Sismic Documentation

Release 1.1.2

Alexandre Decan

Oct 21, 2018

Overview

CHAPTER 1

About

Sismic is a recursive acronym that stands for *Sismic Interactive Statechart Model Interpreter and Checker*.

Statecharts are a well-known visual modeling language for representing the executable behavior of complex reactive event-based systems. Sismic library for Python (version 3.4 or higher) provides a set of tools to define, validate, simulate, execute and test statecharts.

Sismic is mainly developed by Alexandre Decan at the University of Mons and released publicly under the GNU Lesser General Public Licence version 3.0 (LGPLv3).

CHAPTER 2

Features

Sismic provides the following features:

- An easy way to define and to import statecharts, based on the human-friendly YAML markup language
- A statechart interpreter offering a discrete, step-by-step, and fully observable simulation engine
- Built-in support for expressing actions and guards using regular Python code, can be easily extended to other programming languages
- A design-by-contract approach for statecharts: contracts can be specified to express invariants, pre- and post-conditions on states and transitions
- Runtime checking of behavioral properties expressed as statecharts
- Built-in support for behavior-driven development
- Synchronous and asynchronous simulation, in real time or simulated time
- Support for communication between statecharts and co-simulation
- Statechart visualization using PlantUML

Some experimental features are also available as feature branches.

The semantics of the statechart interpreter is based on the specification of the SCXML semantics (with a few exceptions), but can be easily tuned to other semantics. Sismic statecharts provides full support for the majority of the UML 2 statechart concepts:

- simple states, composite states, orthogonal (parallel) states, initial and final states, shallow and deep history states
- state transitions, guarded transitions, automatic (eventless) transitions, internal transitions
- statechart variables and their initialisation
- state entry and exit actions, transition actions
- internal and external parametrized events

2.1 Installation

2.1.1 Using pip

Sismic requires Python >=3.4, and can be installed using `pip` as usual: `pip install sismic`. This will install the latest stable version. Starting from release 1.0.0, Sismic adheres to a semantic versioning scheme.

You can isolate Sismic installation by using virtual environments:

1. Get the tool to create virtual environments: `pip install virtualenv`
2. Create the environment: `virtualenv -p python3.4 env`
3. Jump into: `source env/bin/activate`
4. Install Sismic: `pip install sismic`

The development version can also be installed directly from its git repository: `pip install git+git://github.com/AlexandreDecan/sismic.git`

2.1.2 From GitHub

You can also install Sismic from its repository by cloning it.

1. Get the tool to create virtual environments: `pip install virtualenv`
2. Create the environment: `virtualenv -p python3.4 env`
3. Jump into: `source env/bin/activate`
4. Clone the repository: `git clone https://github.com/AlexandreDecan/sismic`
5. Install Sismic: `pip install .` or `pip install -e .` (editable mode)
6. Install test dependencies: `pip install -r requirements.txt`

Sismic is now available from the root directory. Its code is in the *sismic* repository. The documentation can be built from the *docs* directory using `make html`.

Tests are available both for the code and the documentation:

- `make doctest` in the *docs* directory (documentation tests)
- `python -m pytest tests/` from the root directory (code tests)

2.2 Statecharts definition

2.2.1 About statecharts

Statecharts are a well-known visual language for modeling the executable behavior of complex reactive event-based systems. They were invented in the 1980s by David Harel, and have gained a more widespread adoption since they became part of the UML modeling standard.

Statecharts offer more sophisticated modeling concepts than the more classical state diagrams of finite state machines. For example, they support hierarchical composition of states, orthogonal regions to express parallel execution, guarded transitions, and actions on transitions or states. Different flavours of executable semantics for statecharts have been proposed in the literature and in existing tools.

2.2.2 Defining statecharts in YAML

Because Sismic is supposed to be independent of a particular visual modeling tool, and easy to integrate in other programs without requiring the implementation of a visual notation, statecharts are expressed using YAML, a human-friendly textual notation (the alternative of using something like SCXML was discarded because its notation is too verbose and not really "human-readable").

This section explains how the elements that compose a valid statechart in Sismic can be defined using YAML. If you are not familiar with YAML, have a look at YAML official documentation.

See also:

While statecharts can be defined in YAML, they can be defined in pure Python too. Moreover, `Statechart` instances exhibit several methods to query and manipulate statecharts (e.g.: `rename_state()`, `rotate_transition()`, `copy_from_statechart()`, etc.). Consider looking at `Statechart` API for more information.

See also:

Experimental import/export support for AMOLA specifications of statecharts is available as an extension of Sismic. AMOLA is notably used in ASEME IDE, which can be used to graphically create, edit and visualize statecharts. More information on *Extensions for Sismic*.

Statechart

The root of the YAML file **must** declare a statechart:

```
statechart:
  name: Name of the statechart
  description: Description of the statechart
  root state:
    [...]
```

The *name* and the *root state* keys are mandatory, the *description* is optional. The *root state* key contains a state definition (see below). If specific code needs to be executed during initialization of the statechart, this can be specified using *preamble*. In this example, the code is written in Python.

```
statechart:
  name: statechart containing initialization code
  preamble: x = 1
```

Code can be written on multiple lines as follows:

```
preamble: |
  x = 1
  y = 2
```

States

A statechart must declare a root state. Each state consist of at least a mandatory *name*. Depending on the state type, other optional fields can be declared.

```
statechart:
  name: with state
  root state:
    name: root
```

Entry and exit actions

For each declared state, the optional *on entry* and *on exit* fields can be used to specify the code that has to be executed when entering and leaving the state:

```
- name: s1
  on entry: x += 1
  on exit: |
    x -= 1
    y = 2
```

Final states

A *final state* can be declared by specifying *type: final*:

```
- name: s1
  type: final
```

Shallow and deep history states

History states can be declared as follows:

- *type: shallow history* to declare a *shallow history* state;
- *type: deep history* to declare a *deep history* state.

```
- name: history state
  type: shallow history
```

A history state can optionally declare a default initial memory using *memory*. Importantly, the *memory* value **must** refer to a parent's substate.

```
- name: history state
  type: deep history
  memory: s1
```

See also:

We refer to the semantics of UML for the difference between both types of histories.

Composite states

A state that is neither a final state nor a history state can contain nested states. Such a state is commonly called a *composite state*.

```
- name: composite state
  states:
    - name: nested state 1
    - name: nested state 2
      states:
        - name: nested state 2a
```

A composite state can define its initial state using *initial*.

```yaml
- name: composite state
  initial: nested state 1
  states:
    - name: nested state 1
    - name: nested state 2
      initial: nested state a2
      states:
        - name: nested state 2a
```

Note: Unlike UML, but similarly to SCXML, Sismic does not explicitly represent the concept of *region*. A region is essentially a logical set of nested states, and thus can be viewed as a specialization of a composite state.

Orthogonal states

Orthogonal states (sometimes referred as *parallel states*) allow to specify multiple nested statecharts running in parallel. They must declare their nested states using *parallel states* instead of *states*.

```yaml
statechart:
  name: statechart containing multiple orthogonal states
  initial state:
    name: processes
    parallel states:
      - name: process 1
      - name: process 2
```

Transitions

Transitions between states, compound states and parallel states can be declared with the *transitions* field. Transitions typically specify a target state using the *target* field:

```yaml
- name: state with transitions
  transitions:
    - target: other state
```

Other optional fields can be specified for a transition: a *guard* (a Boolean expression that will be evaluated to determine if the transition can be followed), an *event* (name of the event that will trigger the transition), an *action* (code that will be executed if the transition is processed). Here is a full example of a transition specification:

```yaml
- name: state with an outgoing transition
  transitions:
    - target: some other state
      event: click
      guard: x > 1
      action: print('Hello World!')
```

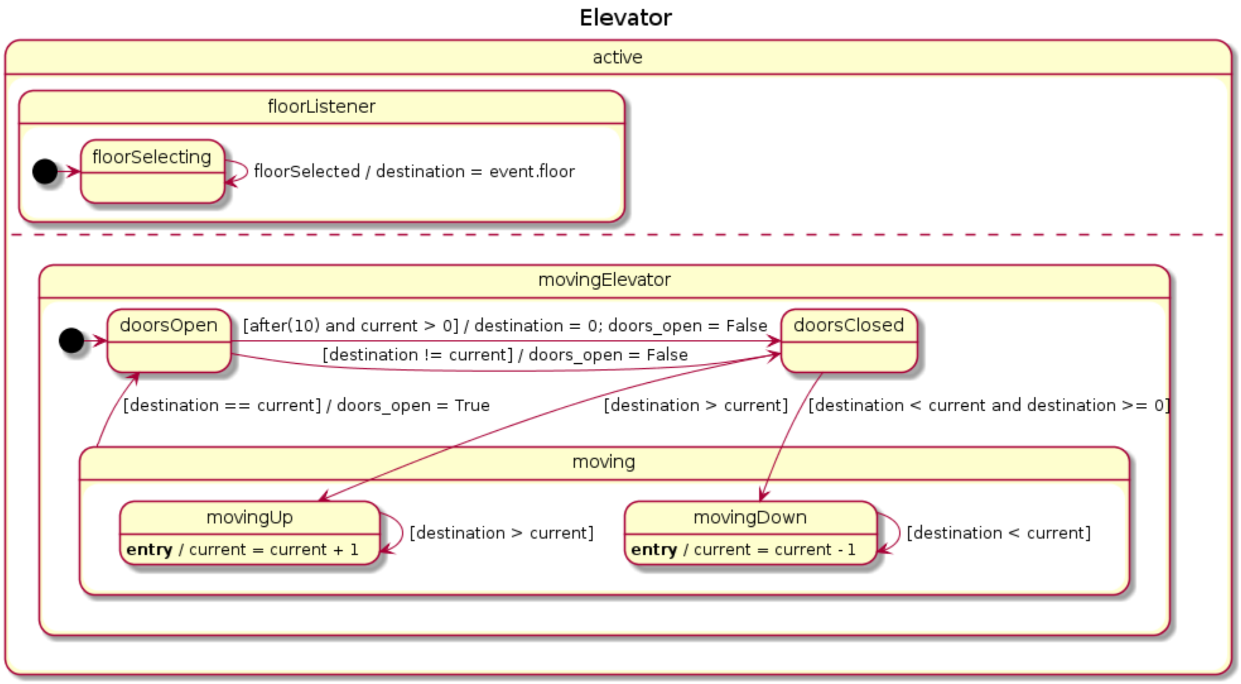
One type of transition, called an *internal transition*, does not require to declare a *target*. Instead, it **must** either define an event or define a guard to determine when it should become active (otherwise, infinite loops would occur during simulation or execution).

Notice that such a transition does not trigger the *on entry* and *on exit* of its state, and can thus be used to model an *internal action*.

Statechart examples

Elevator

The Elevator statechart is one of the running examples in this documentation. Its visual description generated from Sismic using PlantUML looks as follows:



The corresponding YAML description is given below.

```
statechart:
  name: Elevator
  preamble: |
    current = 0
    destination = 0
    doors_open = True
  root state:
    name: active
    parallel states:
      - name: movingElevator
        initial: doorsOpen
        states:
          - name: doorsOpen
            transitions:
              - target: doorsClosed
                guard: destination != current
                action: doors_open = False
              - target: doorsClosed
                guard: after(10) and current > 0
                action: |
                  destination = 0
                  doors_open = False
          - name: doorsClosed
            transitions:
```

(continues on next page)

(continued from previous page)

```
            - target: movingUp
              guard: destination > current
            - target: movingDown
              guard: destination < current and destination >= 0
        - name: moving
          transitions:
            - target: doorsOpen
              guard: destination == current
              action: doors_open = True
          states:
            - name: movingUp
              on entry: current = current + 1
              transitions:
                - target: movingUp
                  guard: destination > current
            - name: movingDown
              on entry: current = current - 1
              transitions:
                - target: movingDown
                  guard: destination < current
    - name: floorListener
      initial: floorSelecting
      states:
        - name: floorSelecting
          transitions:
            - target: floorSelecting
              event: floorSelected
              action: destination = event.floor
```

Other examples

Some other examples can be found in the Git repository of the project, in docs/examples.

2.2.3 Importing and validating statecharts

The *Statechart* class provides several methods to construct, to query and to manipulate a statechart. A YAML definition of a statechart can be easily imported to a *Statechart* instance. The module *sismic.io* provides a convenient loader *import_from_yaml()* which takes a textual YAML definition of a statechart and returns a *Statechart* instance.

sismic.io.**import_from_yaml**(*text=None*, *filepath=None*, ***, *ignore_schema=False*, *ignore_validation=False*)

Import a statechart from a YAML representation (first argument) or a YAML file (filepath argument).

Unless specified, the structure contained in the YAML is validated against a predefined schema (see *sismic.io.SCHEMA*), and the resulting statechart is validated using its *validate()* method.

Parameters

- **text** (Optional[Iterable[str]]) – A YAML text. If not provided, filepath argument has to be provided.
- **filepath** (Optional[str]) – A path to a YAML file.
- **ignore_schema** (bool) – set to *True* to disable yaml validation.

- **ignore_validation** (bool) – set to *True* to disable statechart validation.

Return type Statechart

Returns a *Statechart* instance

For example:

```
from sismic.io import import_from_yaml
from sismic.model import Statechart

with open('examples/elevator/elevator.yaml') as f:
    statechart = import_from_yaml(f)
    assert isinstance(statechart, Statechart)
```

The function also supports importing from a given filepath:

```
statechart = import_from_yaml(filepath='examples/elevator/elevator.yaml')
assert isinstance(statechart, Statechart)
```

The parser performs several checks using statechart's `validate` method. It also does an automatic validation against some kind of schema to prevent erroneous keys. See schema library for more information about the semantics.

```
class SCHEMA:
    contract = {schema.Or('before', 'after', 'always'): schema.Use(str)}

    transition = {
        schema.Optional('target'): schema.Use(str),
        schema.Optional('event'): schema.Use(str),
        schema.Optional('guard'): schema.Use(str),
        schema.Optional('action'): schema.Use(str),
        schema.Optional('contract'): [contract],
    }

    state = dict()  # type: ignore
    state.update({
        'name': schema.Use(str),
        schema.Optional('type'): schema.Or('final', 'shallow history', 'deep history
↪'),
        schema.Optional('on entry'): schema.Use(str),
        schema.Optional('on exit'): schema.Use(str),
        schema.Optional('transitions'): [transition],
        schema.Optional('contract'): [contract],
        schema.Optional('initial'): schema.Use(str),
        schema.Optional('parallel states'): [state],
        schema.Optional('states'): [state],
    })

    statechart = {
        'statechart': {
            'name': schema.Use(str),
            schema.Optional('description'): schema.Use(str),
            schema.Optional('preamble'): schema.Use(str),
            'root state': state,
        }
    }
```

See also:

Consider having a look at the feature branches of Sismic repository to get more information about the various statechart formats that are currently (experimentally) supported but not yet released in Sismic.

2.2.4 Visualising statecharts

Sismic is not bundle with any graphical tool that can be used to edit or even view a statechart. Module *sismic.io* contains routines that can be used to (import and) export statecharts to other format, some of them being used by third-party tools that support visualising (or editing) statecharts.

Notably, module *sismic.io* contains a function *export_to_plantuml()* that export a given statechart to PlantUML, a tool based on graphviz that can automatically render statecharts (to some extent). An online version of PlantUML can be found here.

For example, the elevator statechart can be exported to the following PlantUML file, which in turns can be used to generate the previously given representation of the elevator.

```
@startuml
title Elevator
state "active" as active {
  state "floorListener" as floorListener {
    [*] -right-> floorSelecting
    state "floorSelecting" as floorSelecting {
      floorSelecting --> floorSelecting : floorSelected / destination = event.floor
    }
  }
  --
  state "movingElevator" as movingElevator {
    [*] -right-> doorsOpen
    state "moving" as moving {
      moving --> doorsOpen : [destination == current] / doors_open = True
      state "movingDown" as movingDown {
        movingDown : **entry** / current = current - 1
        movingDown --> movingDown : [destination < current]
      }
      state "movingUp" as movingUp {
        movingUp : **entry** / current = current + 1
        movingUp --> movingUp : [destination > current]
      }
    }
    state "doorsClosed" as doorsClosed {
      doorsClosed --> movingUp : [destination > current]
      doorsClosed --> movingDown : [destination < current and destination >= 0]
    }
    state "doorsOpen" as doorsOpen {
      doorsOpen -right-> doorsClosed : [destination != current] / doors_open = False
      doorsOpen -right-> doorsClosed : [after(10) and current > 0] / destination = 0;
↪doors_open = False
    }
  }
}
@enduml
```

See also:

PlantUML's rendering can be modified to some extent by adjusting the notation used for transitions. By default, `-->` transitions correspond to downward transitions of good length.

A transition can be shortened by using `->` instead of `-->`, and the direction of a transition can be changed by using `-up->`, ``-right->`, ``-down->` or ``-left->`. Both changes can be applied at the same time using *-u->*, `-r->`, ``-d->`` or `-l->`. See PlantUML documentation for more information.

If you have already exported a statechart to PlantUML and made some changes to the direction or length of the transitions, it is likely that you will want to retrieve these changes when you export the (possibly modified) statechart again to PlantUML.

The *export_to_plantuml()* function accepts two optional (mutually exclusive) parameters `based_on` and `based_on_filepath` that can be used to provide an earlier version of a PlantUML text representation (or a path to such a version if `based_on_filepath` is used). This will then be used to incorporate as much as possible the changes made on the transitions.

`sismic.io.`**`export_to_plantuml`**(*statechart*, *filepath=None*, ***, *based_on=None*, *based_on_filepath=None*, *statechart_name=True*, *statechart_description=False*, *statechart_preamble=False*, *state_contracts=False*, *state_action=True*, *transition_contracts=False*, *transition_action=True*)

Export given statechart to plantUML (see http://plantuml/plantuml). If a filepath is provided, also save the output to this file.

Due to the way statecharts are representing, and due to the presence of features that are specific to Sismic, the resulting statechart representation does not include all the informations. For example, final states and history states won't have name, actions and contracts.

If a previously exported representation for the statechart is provided, either as text (based_on parameter) or as a filepath (based_on_filepath parameter), it will attempt to reuse the modifications made to the transitions (their direction and length).

Parameters

- **`statechart`** (`Statechart`) – statechart to export
- **`filepath`** (`Optional[str]`) – save output to given filepath, if provided
- **`based_on`** (`Optional[str]`) – existing representation of the statechart in PlantUML
- **`based_on_filepath`** (`Optional[str]`) – filepath to an existing representation of the statechart in PlantUML
- **`statechart_name`** (`bool`) – include the name of the statechart
- **`statechart_description`** (`bool`) – include the description of the statechart
- **`statechart_preamble`** (`bool`) – include the preamble of the statechart
- **`state_contracts`** (`bool`) – include state contracts
- **`state_action`** (`bool`) – include state actions (on entry, on exit and internal transitions)
- **`transition_contracts`** (`bool`) – include transition contracts
- **`transition_action`** (`bool`) – include actions on transition

Return type `str`

Returns textual representation using plantuml

2.3 Statecharts execution

2.3.1 Statechart semantics

The module *interpreter* contains an *Interpreter* class that interprets a statechart mainly following the SCXML 1.0 semantics. In particular, eventless transitions are processed *before* transitions containing events, internal events are consumed *before* external events, and the simulation follows a inner-first/source-state and run-to-completion semantics.

The main difference between SCXML and Sismic's default interpreter resides in how multiple transitions can be triggered simultaneously. This may occur for transitions in orthogonal/parallel states, or when transitions declaring the same event have guards that are not mutually exclusive.

Simulating the simultaneous triggering of multiple transitions is problematic, since it implies to make a non-deterministic choice on the order in which the transitions must be processed, and on the order in which the source states must the exited and the target states must be entered. The UML 2.5 specification explicitly leaves this issue unresolved, thereby delegating the decision to tool developers:

> "Due to the presence of orthogonal Regions, it is possible that multiple Transitions (in different Regions) can be triggered by the same Event occurrence. The **order in which these Transitions are executed is left undefined**." — UML 2.5 Specification

The SCXML specification addresses the issue by using the *document order* (i.e., the order in which the transitions appear in the SCXML file) as the order in which (non-parallel) transitions should be processed.

> "If multiple matching transitions are present, take the **first in document order**." — SCXML Specification

From our point of view, this solution is not satisfactory. The execution should not depend on the (often arbitrary) order in which items happen to be declared in some document, in particular when there may be many different ways to construct or to import a statechart.

Another statechart tool does not even define any order on the transitions in such situations:

> "Rhapsody detects such cases of nondeterminism during code generation and **does not allow them**. The motivation for this is that the generated code is intended to serve as a final implementation and for most embedded software systems such nondeterminism is not acceptable." — The Rhapsody Semantics of Statecharts

We decide to follow Rhapsody and to raise an error (in fact, a `NonDeterminismError`) if such cases of nondeterminism occur during the execution. Notice that this only concerns multiple transitions in the same composite state, not in parallel states.

When multiple transitions are triggered from within distinct parallel states, the situation is even more intricate. According to the Rhapsody implementation:

> "The order of firing transitions of orthogonal components is not defined, and depends on an arbitrary traversal in the implementation. Also, the actions on the transitions of the orthogonal components are **interleaved in an arbitrary way**." — The Rhapsody Semantics of Statecharts

SCXML circumvents this problem by relying again on the *document order*.

> "enabledTransitions will contain multiple transitions only if a parallel state is active. In that case, we may have one transition selected for each of its children. [. . .] If multiple states are active (i.e., we are in a parallel region), then there may be multiple transitions, one per active atomic state (though some states may not select a transition.) In this case, the transitions are taken **in the document order of the atomic states** that selected them." — SCXML Specification

Again, Sismic does not agree with SCXML on this, and instead defines that multiple orthogonal/parallel transitions should be processed in a decreasing source state depth order. This is perfectly coherent with our aforementioned inner-first/source-state semantics, as "deeper" transitions are processed before "less nested" ones. In case of ties, the lexicographic order of the source state names will prevail.

Note that in an ideal world, orthogonal/parallel regions should be independent, implying that *in principle* such situations should not arise ("*the designer does not rely on any particular order for event instances to be dispatched to the relevant orthogonal regions*", UML specification). In practice, however, it is often desirable to allow such situations.

See also:

Other semantics can be quite easily implemented. For example, the extension *sismic-semantics* already provides support for outer-first/source-state semantics and priority to transitions with event. More information on *Extensions for Sismic*.

2.3.2 Using *Interpreter*

An `Interpreter` instance is constructed upon a `Statechart` instance and an optional callable that returns an `Evaluator`. This callable must accept an interpreter and an initial execution context as input (see *Include code in statecharts*). If not specified, a `PythonEvaluator` will be used. This default evaluator can parse and interpret Python code in statecharts.

Consider the following example:

When an interpreter is built, the statechart is not yet in an initial configuration. To put the statechart in its initial configuration (and to further execute the statechart), call `execute_once()`.

```
print('Before:', interpreter.configuration)

step = interpreter.execute_once()

print('After:', interpreter.configuration)
```

```
Before: []
After: ['active', 'floorListener', 'movingElevator', 'doorsOpen', 'floorSelecting']
```

The method `execute_once()` returns information about what happened during the execution, including the transitions that were processed, the event that was consumed and the sequences of entered and exited states (see *Macro and micro steps* and `sismic.model.MacroStep`).

```
for attribute in ['event', 'transitions', 'entered_states', 'exited_states', 'sent_
↪events']:
    print('{}: {}'.format(attribute, getattr(step, attribute)))
```

```
event: None
transitions: []
entered_states: ['active', ...]
exited_states: []
sent_events: []
```

One can send events to the statechart using its `sismic.interpreter.Interpreter.queue()` method. This method accepts either an `Event` instance, or the name of an event. Multiple events (or names) can be provided at once.

```
from sismic.interpreter import Event

interpreter.queue(Event('click'))
interpreter.execute_once()  # Process the "click" event

interpreter.queue('clack')  # An event name can be provided as well
interpreter.execute_once()  # Process the "clack" event

interpreter.queue('click', 'clack')
interpreter.execute_once()  # Process "click"
interpreter.execute_once()  # Process "clack"
```

For convenience, `queue()` returns the interpreter and thus can be chained:

```
interpreter.queue('click', 'clack').execute_once()
```

Notice that *execute_once()* consumes at most one event at a time. In this example, the *clack* event is not processed.

To process all events **at once**, one can repeatedly call *execute_once()* until it returns a `None` value, meaning that nothing happened during the last call. For instance:

```
while interpreter.execute_once():
  pass
```

For convenience, an interpreter has a *execute()* method that repeatedly call *execute_once()* and that returns a list of its output (a list of *sismic.model.MacroStep*).

```
from sismic.model import MacroStep

interpreter.queue('click', 'clack')

for step in interpreter.execute():
  assert isinstance(step, MacroStep)
```

Notice that a call to *execute()* first computes the list and **then** returns it, meaning that all the steps are already processed when the call returns. As a call to *execute()* could lead to an infinite execution (see for example simple/infinite.yaml), an additional parameter `max_steps` can be specified to limit the number of steps that are computed and executed by the method. By default, this parameter is set to `-1`, meaning there is no limit on the number of calls to *execute_once()*.

```
interpreter.queue('click', 'clack', 'clock')
assert len(interpreter.execute(max_steps=2)) <= 2

# 'clock' is not yet processed
assert len(interpreter.execute()) == 1
```

In these examples, none of *click*, *clack* or *clock* are expected to be received by the statechart. The statechart was not written to react to those events, and thus sending them has no effect on the active configuration.

For convenience, a *Statechart* has an *events_for()* method that returns the list of all possible events that are expected by this statechart.

```
print(elevator.events_for(interpreter.configuration))
```

```
['floorSelected']
```

The *elevator* statechart, the one used for this example, only reacts to *floorSelected* events. Moreover, it assumes that *floorSelected* events have an additional parameter named `floor`. These events are *parametrized* events, and can be created by providing keyword arguments when instanciating *Event*.

```
selecting_floor = Event('floorSelected', floor=1)
```

These parameters can be accessed by action code and guards in the statechart. For example, the *floorSelecting* state of the *elevator* example has a transition `floorSelected / destination = event.floor`.

Executing the statechart will make the elevator reaching first floor:

```
print('Current floor is', interpreter.context['current'])
```

(continues on next page)

(continued from previous page)

```
interpreter.queue(selecting_floor).execute()
print('Current floor is', interpreter.context['current'])
```

```
Current floor is 0
Current floor is 1
```

Notice how we can access to the current values of *internal variables* by use of `context`. This attribute is a mapping between internal variable names and their current value.

2.3.3 Macro and micro steps

An interpreter `execute_once()` (resp. `execute()`) method returns an instance of (resp. a list of) `sismic.model.MacroStep`. A *macro step* corresponds to the process of consuming an event, regardless of the number and the type (eventless or not) of triggered transitions. A macro step also includes every consecutive *stabilization step* (i.e., the steps that are needed to enter nested states, or to switch into the configuration of a history state).

A `MacroStep` exposes the consumed `event` if any, a (possibly empty) list `transitions` of `Transition` instances, and two aggregated ordered sequences of state names, `entered_states` and `exited_states`. In addition, a `MacroStep` exposes a list `sent_events` of events that were fired by the statechart during the considered step. The order of states in those lists determines the order in which their *on entry* and *on exit* actions were processed. As transitions are atomically processed, this means that they could exit a state in `entered_states` that is entered before some state in `exited_states` is exited. The exact order in which states are exited and entered is indirectly available through the `steps` attribute that is a list of all the `MicroStep` that were executed. Each of them contains the states that were exited and entered during its execution, and the a list of events that were sent during the step.

A *micro step* is the smallest, atomic step that a statechart can execute. A `MacroStep` instance thus can be viewed (and is!) an aggregate of `MicroStep` instances.

This way, a complete *run* of a statechart can be summarized as an ordered list of `MacroStep` instances, and details can be obtained using the `MicroStep` list of a `MacroStep`.

2.3.4 Observing the execution

The interpreter is fully observable during its execution. It provides many public and private attributes that can be used to see what happens. In particular:

- The `execute_once()` (resp. `execute()`) method returns an instance of (resp. a list of) `sismic.model.MacroStep`.
- The `log_trace()` function can be used to log all the steps that were processed during the execution of an interpreter. This methods takes an interpreter and returns a (dynamic) list of macro steps.
- The list of active states can be retrieved using `configuration`.
- The context of the execution is available using `context` (see *Include code in statecharts*).
- It is possible to bind a callable that will be called each time an event is sent by the statechart using the `bind()` method of an interpreter (see *Communication between statecharts*).

2.3.5 Anatomy of the interpreter

An `Interpreter` makes use of several *private* methods for its initialization and computations. These methods computes the transition(s) that should be processed, the resulting steps, etc. These methods can be overridden or combined to define variants of statechart semantics.

`Interpreter._select_event(consume=True)`

Return (and consume!) the next available event if any. This method prioritizes internal events over external ones.

Parameters **consume** – Set to False to *not* consume the event.

Return type `Optional[Event]`

Returns An instance of Event or None if no event is available

`Interpreter._select_transitions()`

Select the transitions that could be triggered and the corresponding (optional) event. If automatic transitions (ie. ones without event) are found, return them and do not look for transitions with event. Otherwise, consume next event and return a possibly empty list of transitions that could be fired with this event.

Return type `Tuple[Optional[Event], List[Transition]]`

Returns a couple (event instance, list of *Transition* instances)

`Interpreter._filter_transitions(transitions)`

Given a list of transitions, return a filtered list of transitions with respect to the inner-first/source-state semantic.

Parameters **transitions** (`List[Transition]`) – a list of *Transition* instances

Return type `List[Transition]`

Returns a list of *Transition* instances

`Interpreter._sort_transitions(transitions)`

Given a list of triggered transitions, return a list of transitions in an order that represents the order in which they have to be processed.

Parameters **transitions** (`List[Transition]`) – a list of *Transition* instances

Return type `List[Transition]`

Returns an ordered list of *Transition* instances

Raises **ExecutionError** – In case of non-determinism (*NonDeterminismError*) or conflicting transitions (*ConflictingTransitionsError*).

`Interpreter._create_steps(event, transitions)`

Return a (possibly empty) list of micro steps. Each micro step corresponds to the process of a transition matching given event.

Parameters

- **event** (`Event`) – the event to consider, if any
- **transitions** (`Iterable[Transition]`) – the transitions that should be processed

Return type `List[MicroStep]`

Returns a list of micro steps.

`Interpreter._create_stabilization_step(names)`

Return a stabilization step, ie. a step that lead to a more stable situation for the current statechart. Stabilization means:

- Enter the initial state of a compound state with no active child
- Enter the memory of a history state
- Enter the children of an orthogonal state with no active child
- Empty active configuration if root's child is a final state

Parameters **names** (Iterable[str]) – List of states to consider (usually, the active configuration)

Return type Optional[MicroStep]

Returns A *MicroStep* instance or *None* if this statechart can not be more stabilized

Interpreter.**_apply_step**(*step*)
Apply given *MicroStep* on this statechart

Parameters **step** (MicroStep) – *MicroStep* instance

Return type MicroStep

Returns a new MicroStep, completed with sent events

These methods are all used (even indirectly) by *execute_once*.

See also:

Consider looking at the source of *execute_once* to understand how these methods are related and organized.

2.4 Include code in statecharts

2.4.1 Python code evaluator

A statechart can specify code that needs to be executed under some circumstances. For example, the *preamble* of a statechart, the *guard* or *action* of a transition or the *on entry* and *on exit* of a state may all contain code.

In Sismic, these pieces of code can be evaluated and executed by *Evaluator* instances. By default, when an interpreter is created, a *PythonEvaluator* is created and allows the interpreter to evaluate and execute Python code contained in a statechart.

Alternatively, a *DummyEvaluator* that always evaluates conditions to True and silently ignores actions can be used, but is clearly of less interest.

In the following, we will implicitly assume that the code evaluator is an instance of *PythonEvaluator*.

2.4.2 Context of the Python code evaluator

When a code evaluator is created or provided to an interpreter, its context is exposed through the context attribute of the interpreter. The context of an evaluator is a mapping between variable names and their values. When a piece of code contained in a statechart has to be evaluated or executed, the context of the evaluator is used to populate the local and global variables that are available for this piece of code.

As an example, consider the following partial statechart definition.

```
statechart:
  # ...
  preamble: |
    x = 1
    y = 0
  root state:
    name: s1
    on entry: x += 1
```

When an interpreter is created for this statechart, its preamble is executed and the context of the code evaluator is populated with `{'x': 1, 'y': 0}`. When the statechart is further executed (initialized), and its root state *s1* is entered, the code `x += 1` contained in the `on entry` field of *s1* is then executed in this context. After execution, the context is `{'x': 2, 'y': 0}`.

The default code evaluator has a global context that is always exposed when a piece of code has to be evaluated or executed. When a `PythonEvaluator` instance is initialized, an initial context can be specified. For convenience, the initial context can be directly provided to the constructor of an `Interpreter`.

It should be noticed that the initial context is set *before* executing the preamble of a statechart. While this should be expected, it has the direct consequence that if a variable defined in the initial context is also defined by the preamble, the latter will override its value, as illustrated by the following example:

```
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter
import math as my_favorite_module

yaml = """statechart:
  name: example
  preamble:
    x = 1
  root state:
    name: s
"""

statechart = import_from_yaml(yaml)
context = {
    'x': 2,
    'math': my_favorite_module
}

interpreter = Interpreter(statechart, initial_context=context)

print(interpreter.context['x'])
```

```
1
```

In this example, the value of `x` is eventually set to `1`. While the initial context provided to the interpreter defined the value of `x` to `2`, the code contained in the preamble overrode its value. If you want to make use of the initial context to somehow *parametrize* the execution of the statechart, while still providing *default* values for these parameters, you should check the existence of the variables before setting their values. This can be done as follows:

```
if not 'x' in locals():
    x = 1
```

or equivalently,

```
x = locals().get('x', 1)
```

> **Warning:** Under the hood, a Python evaluator makes use of `eval()` and `exec()` with global and local contexts. This can lead to some *weird* issues with variable scope (as in list comprehensions or lambda's). See this question on Stackoverflow for more information.

2.4.3 Predefined variables and functions

When a piece of code is evaluated or executed, the default Python code evaluator enriches its local context with several predefined variables and functions. These predefined objects depend on the situation triggering a code evaluation or a code execution (entry or exit actions, guard evaluation, transition action, . . .).

These entries are covered in the docstring of a `PythonEvaluator`:

class `sismic.code.`**`PythonEvaluator`**(*interpreter=None*, *, *initial_context=None*)

A code evaluator that understands Python.

Depending on the method that is called, the context can expose additional values:

- **On both code execution and code evaluation:**
  - A *time: float* value that represents the current time exposed by the interpreter.
  - An *active(name: str) -> bool* Boolean function that takes a state name and return *True* if and only if this state is currently active, ie. it is in the active configuration of the `Interpreter` instance that makes use of this evaluator.
- **On code execution:**
  - A *send(name: str, **kwargs) -> None* function that takes an event name and additional keyword parameters and raises an internal event with it.
  - If the code is related to a transition, the *event: Event* that fires the transition is exposed.
- **On guard or contract evaluation:**
  - If the code is related to a transition, the *event: Event* that fires the transition is exposed.
- **On guard or contract (except preconditions) evaluation:**
  - An *after(sec: float) -> bool* Boolean function that returns *True* if and only if the source state was entered more than *sec* seconds ago. The time is evaluated according to Interpreter's clock.
  - A *idle(sec: float) -> bool* Boolean function that returns *True* if and only if the source state did not fire a transition for more than *sec* ago. The time is evaluated according to Interpreter's clock.
- **On contract (except preconditions) evaluation:**
  - A variable *__old__* that has an attribute *x* for every *x* in the context when either the state was entered (if the condition involves a state) or the transition was processed (if the condition involves a transition). The value of *__old__.x* is a shallow copy of *x* at that time.
- **On contract evaluation:**
  - A *sent(name: str) -> bool* function that takes an event name and return True if an event with the same name was sent during the current step.
  - A *received(name: str) -> bool* function that takes an event name and return True if an event with the same name is currently processed in this step.

If an exception occurred while executing or evaluating a piece of code, it is propagated by the evaluator.

Parameters

- **`interpreter`** – the interpreter that will use this evaluator, is expected to be an *Interpreter* instance
- **`initial_context`** (`Optional[Mapping[str, Any]]`) – a dictionary that will be used as *__locals__*

2.4.4 Anatomy of a code evaluator

An *Evaluator* subclass must at lest implement the following methods and attributes:

Evaluator.**_evaluate_code**(*code*, *, *additional_context=None*)

Generic method to evaluate a piece of code. This method is a fallback if one of the other evaluate_* methods is not overridden.

Parameters

- **code** (str) – code to evaluate
- **additional_context** (Optional[Mapping[str, Any]]) – an optional additional context

Return type bool

Returns truth value of *code*

Evaluator.**_execute_code**(*code*, *, *additional_context=None*)

Generic method to execute a piece of code. This method is a fallback if one of the other execute_* methods is not overridden.

Parameters

- **code** (str) – code to execute
- **additional_context** (Optional[Mapping[str, Any]]) – an optional additional context

Return type List[Event]

Returns a list of sent events

Evaluator.**context**

The context of this evaluator. A context is a dict-like mapping between variables and values that is expected to be exposed when the code is evaluated.

Return type Mapping[str, Any]

Note: None of those two methods are actually called by the interpreter during the execution of a statechart. These methods are *fallback methods* that are used by other methods that are implicitly called depending on what is currently being processed in the statechart. The documentation of *Evaluator* covers this:

---

class sismic.code.**Evaluator**(*interpreter=None*, *, *initial_context=None*)

Abstract base class for any evaluator.

An instance of this class defines what can be done with piece of codes contained in a statechart (condition, action, etc.).

Notice that the execute_* methods are called at each step, even if there is no code to execute. This allows the evaluator to keep track of the states that are entered or exited, and of the transitions that are processed.

Parameters

- **interpreter** – the interpreter that will use this evaluator, is expected to be an *Interpreter* instance
- **initial_context** (Optional[Mapping[str, Any]]) – an optional dictionary to populate the context

evaluate_guard(*transition*, *event=None*)

Evaluate the guard for given transition.

Parameters

- **transition** (Transition) – the considered transition
- **event** (Optional[Event]) – instance of *Event* if any

Return type Optional[bool]

Returns truth value of *code*

evaluate_invariants(*obj*, *event=None*)

Evaluate the invariants for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterable[str]

Returns list of unsatisfied conditions

evaluate_postconditions(*obj*, *event=None*)

Evaluate the postconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterable[str]

Returns list of unsatisfied conditions

evaluate_preconditions(*obj*, *event=None*)

Evaluate the preconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterable[str]

Returns list of unsatisfied conditions

execute_action(*transition*, *event=None*)

Execute the action for given transition. This method is called for every transition that is processed, even those with no *action*.

Parameters

- **transition** (Transition) – the considered transition
- **event** (Optional[Event]) – instance of *Event* if any

Return type List[Event]

Returns a list of sent events

execute_on_entry(*state*)
Execute the on entry action for given state. This method is called for every state that is entered, even those with no *on_entry*.

Parameters **state** (StateMixin) – the considered state

Return type List[Event]

Returns a list of sent events

execute_on_exit(*state*)
Execute the on exit action for given state. This method is called for every state that is exited, even those with no *on_exit*.

Parameters **state** (StateMixin) – the considered state

Return type List[Event]

Returns a list of sent events

execute_statechart(*statechart*)
Execute the initial code of a statechart. This method is called at the very beginning of the execution.

Parameters **statechart** (Statechart) – statechart to consider

on_step_starts(*event=None*)
Called each time the interpreter starts a macro step.

Parameters **event** (Optional[Event]) – Optional processed event

Return type None

2.5 Design by Contract for statecharts

2.5.1 About Design by Contract

Design by Contract (DbC) was introduced by Bertrand Meyer and popularised through his object-oriented Eiffel programming language. Several other programming languages also provide support for DbC. The main idea is that the specification of a software component (e.g., a method, function or class) is extended with a so-called *contract* that needs to be respected when using this component. Typically, the contract is expressed in terms of preconditions, postconditions and invariants.

> Design by contract (DbC), also known as contract programming, programming by contract and design-by-contract programming, is an approach for designing software. It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with **preconditions, postconditions and invariants**. These specifications are referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts. — Wikipedia

2.5.2 DbC for statechart models

While DbC has gained some amount of acceptance at the programming level, there is hardly any support for it at the modeling level.

Sismic aims to change this, by integrating support for Design by Contract for statecharts. The basic idea is that contracts can be defined on statechart componnents (states or transitions), by specifying preconditions, postconditions, and invariants on them. At runtime, Sismic will verify the conditions specified by the contracts. If a condition is not satisfied, a `ContractError` will be raised. More specifically, one of the following 4 error types wil be raised: `PreconditionError`, `PostconditionError`, or `InvariantError`.

Contracts can be specified for any state contained in the statechart, and for any transition contained in the statechart. A state contract can contain preconditions, postconditions, and/or invariants. The semantics for evaluating a contract is as follows:

- **For states:**
  - state preconditions are checked before the state is entered (i.e., **before** executing *on entry*), in the order of occurrence of the preconditions.
  - state postconditions are checked after the state is exited (i.e., **after** executing *on exit*), in the order of occurrence of the postconditions.
  - state invariants are checked at the end of each *macro step*, in the order of occurrence of the invariants. The state must be in the active configuration.
- **For transitions:**
  - the preconditions are checked before starting the process of the transition (and **before** executing the optional transition action).
  - the postconditions are checked after finishing the process of the transition (and **after** executing the optional transition action).
  - the invariants are checked twice: one before starting and a second time after finishing the process of the transition.

2.5.3 Defining contracts in YAML

Contracts can easily be added to the YAML definition of a statechart (see *Defining statecharts in YAML*) through the use of the *contract* property. Preconditions, postconditions, and invariants are defined as nested items of the *contract* property. The name of these optional contractual conditions is respectively *before* (for preconditions), *after* (for postconditions), and *always* (for invariants):

```
contract:
 - before: ...
 - after: ...
 - always: ...
```

Obviously, more than one condition of each type can be specified:

```
contract:
 - before: ...
 - before: ...
 - before: ...
 - after: ...
```

A condition is an expression that will be evaluated by an `Evaluator` instance (see *Include code in statecharts*).

```
contract:
 - before: x > 0
 - before: y > 0
 - after: x + y == 0
 - always: x + y >= 0
```

Here is an example of a contracts defined at state level:

```
statechart:
  name: example
  root state:
    name: root
    contract:
     - always: x >= 0
     - always: not active('other state') or x > 0
```

If the default *PythonEvaluator* is used, it is possible to refer to the old value of some variable used in the statechart, by prepending `__old__`. This is particularly useful when specifying postconditions and invariants:

```
contract:
  always: d > __old__.d
  after: (x - __old__.x) < d
```

See the documentation of *PythonEvaluator* for more information.

2.5.4 Executing statecharts containing contracts

The execution of a statechart that contains contracts does not essentially differ from the execution of a statechart that does not. The only difference is that conditions of each contract are checked at runtime (as explained above) and may raise a subclass of *ContractError*.

```
from sismic.interpreter import Interpreter, Event
from sismic.io import import_from_yaml

statechart = import_from_yaml(filepath='examples/elevator/elevator_contract.yaml')

# Make the run fails
statechart.state_for('movingUp').preconditions[0] = 'current > destination'

interpreter = Interpreter(statechart)
interpreter.queue(Event('floorSelected', floor=4))
interpreter.execute()
```

Here we manually changed one of the preconditions such that it failed at runtime. The exception displays some relevant information to help debug:

```
Traceback (most recent call last):
 ...
sismic.exceptions.PreconditionError: PreconditionError
Object: BasicState('movingUp')
Assertion: current > destination
Configuration: ['active', 'floorListener', 'movingElevator', 'floorSelecting', 'moving
↪']
Step: MicroStep(transition=Transition('doorsClosed', 'movingUp', event=None), entered_
↪states=['moving', 'movingUp'], exited_states=['doorsClosed'])
Context:
 - current = 0
 - destination = 4
 - doors_open = False
```

If you do not want the execution to be interrupted by such exceptions, you can set the `ignore_contract` parameter to `True` when constructing an `Interpreter`. This way, no contract checking will be done during the execution.

2.6 Monitoring properties

2.6.1 About runtime verification

Like any executable software artefacts, statecharts can and should be tested during their development.

One possible appproach is to test the execution of a statechart *by hand*, writing unit tests or BDD tests. The Sismic interpreter stores and returns several values that can be inspected during the execution, including the active configuration, the list of entered or exited states, etc. The functional tests in *tests/test_interpreter.py* on the GitHub repository are several examples of this kind of tests.

Another key feature of Sismic's interpreter is its support for monitoring properties at runtime, not only contracts. To avoid a statechart designer needing to learn a different language for expressing such properties, these properties are expressed using the statechart notation. These properties are then called *property statecharts*

2.6.2 Using statecharts to express properties

Property statecharts can be used to express functional properties of the intended behaviour in terms of the events that are consumed or sent, or in terms of the states that are entered or exited by a statechart. When a statechart is executed by Sismic, specific meta-events are created based on the events that are sent or consumed, the states that are entered of exited, etc. When the statechart being monitored is executed, the meta-events are propagated to all associated property statecharts. The property statecharts will look for property violations based on those meta-events, following a *fail fast* approach: they will report a failure as soon as the monitored behavior leads to a final state of the property statechart.

Due to the meta-events being considered and the "fail-fast" approach adopted by Sismic for their verification, property statecharts are mainly intended to check for the presence of undesirable behavior (safety properties), i.e., properties that can be checked on a (finite) prefix of a (possibly infinite) execution trace. While it is technically possible to use property statecharts to express liveliness properties (something desirable *eventually* happens), this would require additional code for their verification since liveliness properties are not supported "as is" by Sismic.

During the execution of a statechart, several meta-events are created depending on what happens in the statechart being executed. Those meta-events are automatically send to any previously bound property statechart.

To bind a property statechart to an interpreter, it suffices to provide the property statechart as a parameter of the `bind_property_statechart()` method of an interpreter. This method accepts either a `Statechart` or an `Interpreter` instance.

When a property statechart is bound to an interpreter, its internal clock (the `time` attribute) is automatically synchronised with the one of the interpreter.

If a property statechart reaches a final state during its execution, then the property is considered as not satisfied, and a `PropertyStatechartError` is raised. This exception provides access to the interpreter that executed the property, the active configuration of statechart being executed, the latest executed `MacroStep` and the current context of the interpreter.

2.6.3 Meta-events generated by the interpreter

The complete list of `MetaEvent` that are created by the interpreter is described in the documentation of the `bind_property_statechart()` method:

`Interpreter.`**`bind_property_statechart`**`(`*statechart_or_interpreter*`)`

Bind a property statechart to the current interpreter. A property statechart receives meta-events from the current interpreter depending on what happens:

- *step started*: when a macro step starts.

- *step ended*: when a macro step ends.
- *event consumed*: when an event is consumed. The consumed event is exposed through the `event` attribute.
- *event sent*: when an event is sent. The sent event is exposed through the `event` attribute.
- *state exited*: when a state is exited. The exited state is exposed through the `state` attribute.
- *state entered*: when a state is entered. The entered state is exposed through the `state` attribute.
- *transition processed*: when a transition is processed. The source state, target state and the event are exposed respectively through the `source`, `target` and `event` attribute.

The internal clock of all property statecharts will be synced with the one of the current interpreter. As soon as a property statechart reaches a final state, a `PropertyStatechartError` will be raised, implicitly meaning that the property expressed by the corresponding property statechart is not satisfied.

Parameters **statechart_or_interpreter** (`Union[Statechart, Interpreter]`) – A property statechart or an interpreter of a property statechart.

Return type None

2.6.4 Examples of property statecharts

7th floor is never reached

This property statechart ensures that the 7th floor is never reached. It stores the current floor based on the number of times the elevator goes up and goes down.

```
statechart:
  name: Test that the elevator never reachs 7th floor
  preamble: floor = 0
  root state:
    name: root
    initial: standing
    states:
      - name: standing
        transitions:
          - event: state entered
            guard: event.state == 'moving'
            target: moving
          - guard: floor == 7
            target: fail
      - name: moving
        transitions:
          - event: state entered
            guard: event.state == 'movingUp'
            action: floor += 1
          - event: state entered
            guard: event.state == 'movingDown'
            action: floor -= 1
          - event: state exited
            guard: event.state == 'moving'
            target: standing
      - name: fail
        type: final
```

Elevator moves after 10 seconds

This property statechart checks that the elevator automatically moves after some idle time if it is not on the ground floor. The test sets a timeout of 12 seconds, but it should work for any number strictly greater than 10 seconds.

```
statechart:
  name: Test that the elevator goes to ground floor after 10 seconds (timeout set to␣
↪12 seconds)
  preamble: floor = 0
  root state:
    name: root
    initial: active
    states:
      - name: active
        parallel states:
          - name: guess floor
            transitions:
              - event: state entered
                guard: event.state == 'movingUp'
                action: floor += 1
              - event: state entered
                guard: event.state == 'movingDown'
                action: floor -= 1
          - name: check timeout
            initial: standing
            states:
              - name: standing
                transitions:
                  - event: state entered
                    guard: event.state == 'moving'
                    target: moving
                  - guard: after(12) and floor != 0
                    target: timeout
              - name: moving
                transitions:
                  - event: state exited
                    guard: event.state == 'moving'
                    target: standing
              - name: timeout
                type: final
```

Heating does not start if door is opened

This property statechart checks that the heating of a microwave could not start if the door is currently opened.

```
statechart:
  name: Heating does not start if door is opened
  root state:
    name: root
    initial: door is closed
    states:
      - name: door is closed
        transitions:
          - target: door is opened
            event: event consumed
            guard: event.event.name == 'door_opened'
```

(continues on next page)

(continued from previous page)

```
      - name: door is opened
        transitions:
          - target: door is closed
            event: event consumed
            guard: event.event.name == 'door_closed'
          - target: failure
            event: event sent
            guard: event.event.name == 'heating_on'
      - name: failure
        type: final
```

Heating must stop when door is opened

This property statechart ensures that the heating should quickly stop when the door is open while cooking occurs.

```
statechart:
  name: Test that the elevator never reachs 7th floor
  preamble: floor = 0
  root state:
    name: root
    initial: standing
    states:
      - name: standing
        transitions:
          - event: state entered
            guard: event.state == 'moving'
            target: moving
          - guard: floor == 7
            target: fail
      - name: moving
        transitions:
          - event: state entered
            guard: event.state == 'movingUp'
            action: floor += 1
          - event: state entered
            guard: event.state == 'movingDown'
            action: floor -= 1
          - event: state exited
            guard: event.state == 'moving'
            target: standing
      - name: fail
        type: final
```

2.7 Behavior-Driven Development

2.7.1 About Behavior-Driven Development

This introduction is inspired by the documentation of Behave, a Python library for Behavior-Driven Development (BDD). BDD is an agile software development technique that encourages collaboration between developers, QA and non-technical or business participants in a software project. It was originally named in 2003 by Dan North as a response to test-driven development (TDD), including acceptance test or customer test driven development practices as found in extreme programming.

BDD focuses on obtaining a clear understanding of desired software behavior through discussion with stakeholders. It extends TDD by writing test cases in a natural language that non-programmers can read. Behavior-driven developers use their native language in combination with the language of domain-driven design to describe the purpose and benefit of their code. This allows developers to focus on why the code should be created, rather than the technical details, and minimizes translation between the technical language in which the code is written and the domain language spoken by the business, users, stakeholders, project management, etc.

2.7.2 The Gherkin language

The Gherkin language is a business readable, domain specific language created to support behavior descriptions in BDD. It lets you describe software's behaviour without the need to know its implementation details. Gherkin allows the user to describe a software feature or part of a feature by means of representative scenarios of expected outcomes. Like YAML or Python, Gherkin aims to be a human-readable line-oriented language.

Here is an example of a feature and scenario description with Gherkin, describing part of the intended behaviour of the Unix ls command:

```
Feature: ls
In order to see the directory structure
As a UNIX user
I need to be able to list the current directory's contents

Scenario: List 2 files in a directory
    Given I am in a directory "test"
    And I have a file named "foo"
    And I have a file named "bar"
    When I run "ls"
    Then I should get:
        """
        bar
        foo
        """
```

As can be seen above, Gherkin files should be written using natural language - ideally by the non-technical business participants in the software project. Such feature files serve two purposes: documentation and automated tests. Using one of the available Gherkin parsers, it is possible to execute the described scenarios and check the expected outcomes.

See also:

A quite complete overview of the Gherkin language is available here.

2.7.3 Sismic support for BDD

Since statecharts are executable pieces of software, it is desirable for statechart users to be able to describe the intended behavior in terms of feature and scenario descriptions. While it is possible to manually integrate the BDD process with any library or software, Sismic is bundled with a command-line utility `sismic-bdd` (or `python -m sismic.bdd`) that automates the integration of BDD.

Sismic support for BDD relies on Behave, a Python library for BDD with full support of the Gherkin language.

As an illustrative example, let us define the desired behavior of our elevator statechart. We first create a feature file that contains several scenarios of interest. By convention, this file has the extension *.feature*, but this is not mandatory. The example illustrates that Sismic provides a set of predefined steps (e.g., *given*, *when*, *then*) to describe common statechart behavior without having to write a single line of Python code.

```
Feature: Elevator

  Scenario: Elevator starts on ground floor
    When I do nothing
    Then variable current equals 0
    And variable destination equals 0

  Scenario: Elevator can move to 7th floor
    When I send event floorSelected with floor=7
    Then variable current equals 7

  Scenario: Elevator can move to 4th floor
    When I send event floorSelected
      | parameter  | value |
      | floor      | 4     |
      | dummy      | None  |
    Then variable current equals 4

  Scenario: Elevator reaches ground floor after 10 seconds
    When I reproduce "Elevator can move to 7th floor"
    Then variable current equals 7
    When I wait 10 seconds
    Then variable current equals 0
    # Example using another step:
    And expression "current == 0" holds

  Scenario Outline: Elevator can reach floor from 0 to 5
    When I send event floorSelected with floor=<floor>
    Then variable current equals <floor>

    Examples:
      | floor |
      | 0     |
      | 1     |
      | 2     |
      | 3     |
      | 4     |
      | 5     |
```

Let us save this file as *elevator.feature* in the same directory as the statechart description, *elevator.yaml*. We can then instruct `sismic-bdd` to run on this statechart the scenarios described in the feature file:

```
sismic-bdd elevator.yaml --features elevator.feature
```

Under the hood, `sismic-bdd` will create a temporary directory where all the files required to execute Behave are put. It also makes available a list of predefined *given*, *when*, and *then* steps and sets up many hooks that are required to integrate Sismic and Behave.

Note: Module `sismic.bdd` exposes a *execute_bdd()* function that is internally used by `sismic-bdd` CLI, and that can be used if programmatic access to these features is required.

When `sismic-bdd` is executed, it will somehow translate the feature file into executable code, compute the outcomes of the scenarios, check whether they match what is expected, and display as summary of all executed scenarios and encountered errors:

```
[...]

1 feature passed, 0 failed, 0 skipped
10 scenarios passed, 0 failed, 0 skipped
22 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.027s
```

The `sismic-bdd` command-line interface accepts several other parameters:

```
usage: sismic-bdd [-h] --features features [features ...]
                  [--steps steps [steps ...]]
                  [--properties properties [properties ...]] [--show-steps]
                  [--debug-on-error]
                  statechart

Command-line utility to execute Gherkin feature files using Behave. Extra parameters␣
↪will be passed to Behave.

positional arguments:
  statechart            A YAML file describing a statechart

optional arguments:
  -h, --help            show this help message and exit
  --features features [features ...]
                        A list of files containing features
  --steps steps [steps ...]
                        A list of files containing steps implementation
  --properties properties [properties ...]
                        A list of filepaths pointing to YAML property
                        statecharts. They will be checked at runtime following
                        a fail fast approach.
  --show-steps          Display a list of available steps (equivalent to
                        Behave's --steps parameter
  --debug-on-error      Drop in a debugger in case of step failure (ipdb if
                        available)
```

Additionally, any extra parameter provided to `sismic-bdd` will be passed to Behave. See command-line parameters of Behave for more information.

2.7.4 Predefined steps

In order to be able to execute scenarios, a Python developer needs to write code defining the mapping from the actions and assertions expressed as natural language sentences in the scenarios (using specific keywords such as *given*, *when* or *then*) to Python code that manipulates the statechart. To facilitate the implementation of this mapping, Sismic provides a set of predefined statechart-specific steps.

By convention, steps starting with *given* or *when* correspond to actions that must be applied on the statechart, while steps starting with *then* correspond to assertions about the execution or the current state of the statechart. More precisely, (1) all *given* or *when* steps implicitly call the `execute()` method of the underlying interpreter, (2) all *when* steps capture the output of these calls, and (3) we developed all predefined *then* steps to assert things based on the captured output (implying that only the steps that start with *when* will be monitored in practice).

"Given" and "when" steps

Given/when I send event {name}

This step queues an event with provided name.

Given/when I send event {name} with {parameter}={value}

This step queues an event with provided name and parameter. More than one parameter can be specified when using Gherkin tables, as follows:

```
Scenario: Elevator can move to 4th floor
  When I send event floorSelected
    | parameter  | value |
    | floor      | 4     |
    | dummy      | None  |
```

Given/when I wait {seconds:g} seconds

Given/when I wait {seconds:g} second

These steps increase the internal clock of the interpreter.

Given/when I do nothing

This step does nothing. It's main usage is when assertions using *then* steps are written as first steps of a scenario. As they require a *when* step to be present, use "when I do nothing".

Given/when I reproduce "{scenario}"

This step reproduces all the *given* and *when* steps that are contained in provided scenario. When this step is prefixed with *given* (resp. *when*), the steps of the provided scenario will be reproduced using *given* (resp. *when*).

```
Scenario: Elevator can move to 7th floor
  When I send event floorSelected with floor=7
  Then variable current equals 7

Scenario: Elevator reaches ground floor after 10 seconds
  When I reproduce "Elevator can move to 7th floor"
  Then variable current equals 7
  When I wait 10 seconds
  Then variable current equals 0
```

Given/when I repeat "{step}" {repeat:d} times

This step repeats given step several times. The text of the step must be provided without its keyword, and will be executed using the current keyword (*given* or *when*).

"Then" steps

Then state {name} is entered

Then state {name} is not entered

Then state {name} is exited

Then state {name} is not exited

These steps assert that a state with provided name was respectively entered, not entered, exited, not exited.

Then state {name} is active

Then state {name} is not active

These steps assert that a state with provided name is (not) in the active configuration of the statechart.

Then event {name} is fired

Then event {name} is fired with {parameter}={value}

These steps assert that an event with provided name was sent. Additional parameters can be provided using Gherkin tables.

Then event {name} is not fired

This step asserts that no event with provided name was sent.

Then no event is fired

This step asserts that no event was fired.

Then variable {variable} equals {value}

This step asserts that the context of the statechart has a variable with a given name and a given value.

Then variable {variable} does not equal {value}

This step asserts that the context of a statechart has a variable with a given name, but a value different than the one that is provided.

Then expression "{expression}" holds

Then expression "{expression}" does not hold

These steps assert that given expression holds (does not hold). The expression will be evaluated by the underlying code evaluator (a `PythonEvaluator` by default) using the current context.

Then statechart is in a final configuration

Then statechart is not in a final configuration

These steps assert that the statechart is (not) in a final configuration.

2.7.5 Implementing new steps

While the steps that are already predefined should be sufficient to manipulate the statechart, it is more intuitive to use domain-specific steps to write scenarios. For example, if the statechart being tested encodes the behavior of a microwave oven, the domain-specific step "Given I open the door" corresponds to the action of sending an event `door_opened` to the statechart, and is more intuitive to use when writing scenarios.

Consider the following scenarios expressed using a domain-specific language:

```
Feature: Cook food

  Scenario: Cook food
    Given I open the door
    And I place an item in the oven
    And I close the door
    And I press increase timer button 5 times
    And I press increase power button
    When I press start button
    Then heating turns on

  Scenario: No heating when door is not closed
    Given I reproduce "Cook food"
    And I open the door
    When I press start button
    Then heating does not turn on
```

(continues on next page)

(continued from previous page)

```
Scenario: Opening door interrupts heating
  Given I reproduce "Cook food"
  And 3 seconds elapsed
  When I open the door
  Then heating turns off

Scenario: Lamp is on when door is open
  When I open the door
  Then lamp turns on
  When I close the door
  Then lamp turns off

Scenario: Lamp is on while cooking
  When I reproduce "Cook food"
  Then lamp turns on

Scenario: Cooking can be stopped stop
  Given I reproduce "Cook food"
  When 2 seconds elapsed
  Then variable timer equals 3
  When I press stop button
  Then variable timer equals 0
  And heating turns off
```

The mapping from domain-specific step "Given I open the door" to the action of sending a door opened event to the statechart could be defined using plain Python code, by defining a new step following Python Step Implementations of Behave.

```
from behave import given, when

@given('I open the door')
@when('I open the door')
def opening_door(context):
    context.interpreter.queue('door_opened')
```

For convenience, the `context` parameter automatically provided by Behave at runtime exposes three Sismic-specific attributes, namely `interpreter`, `trace` and `monitored_trace`. The first one corresponds to the interpreter being executed, the second one is a list of all executed macro steps, and the third one is list of executed macro steps restricted to the ones that were performed during the execution of the previous block of *when* steps.

However, this domain-specific step can also be implemented more easily as an alias of predefined step "Given I send event door_opened". As we believe that most of the domain-specific steps are just aliases or combinations of predefined steps, Sismic provides two convenient helpers to map new steps to predefined ones:

`sismic.bdd.`**`map_action`**(*step_text*, *existing_step_or_steps*)

Map new "given"/"when" steps to one or many existing one(s). Parameters are propagated to the original step(s) as well, as expected.

Examples:

- map_action('I open door', 'I send event open_door')
- map_action('Event {name} has to be sent', 'I send event {name}')
- map_action('I do two things', ['First thing to do', 'Second thing to do'])

Parameters

- **step_text** (str) – Text of the new step, without the "given" or "when" keyword.
- **existing_step_or_steps** (Union[str, List[str]]) – existing step, without the "given" or "when" keyword. Could be a list of steps.

Return type None

sismic.bdd.**map_assertion** (*step_text*, *existing_step_or_steps*)

Map a new "then" step to one or many existing one(s). Parameters are propagated to the original step(s) as well, as expected.

map_assertion('door is open', 'state door open is active') map_assertion('{x} seconds elapsed', 'I wait for {x} seconds') map_assertion('assert two things', ['first thing to assert', 'second thing to assert'])

Parameters

- **step_text** (str) – Text of the new step, without the "then" keyword.
- **existing_step_or_steps** (Union[str, List[str]]) – existing step, without "then" keyword. Could be a list of steps.

Return type None

Using these helpers, one can easily implement the domain-specific steps of our example:

```
from sismic.bdd import map_action, map_assertion


map_action('I open the door', 'I send event door_opened')
map_action('I close the door', 'I send event door_closed')
map_action('I place an item in the oven', 'I send event item_placed')
map_action('I press increase timer button {time} times', 'I repeat "I send event␣
↪timer_inc" {time} times')
map_action('I press increase power button', 'I send event power_inc')
map_action('I press start button', 'I send event cooking_start')
map_action('I press stop button', 'I send event cooking_stop')
map_action('{tick} seconds elapsed', 'I repeat "I send event timer_tick" {tick} times
↪')

map_assertion('Heating turns on', 'Event heating_on is fired')
map_assertion('Heating does not turn on', 'Event heating_on is not fired')
map_assertion('heating turns off', 'Event heating_off is fired')
map_assertion('lamp turns on', 'Event lamp_switch_on is fired')
map_assertion('lamp turns off', 'Event lamp_switch_off is fired')
```

Assuming that the features are defined in `heating.feature`, these steps in `steps.py`, and the microwave in `microwave.yaml`, then `sismic-bdd` can be used as follows:

```
$ sismic-bdd microwave.yaml --steps steps.py --features heating.feature

Feature: Cook food # heating.feature:1

[...]

1 feature passed, 0 failed, 0 skipped
5 scenarios passed, 0 failed, 0 skipped
21 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.040s
```

2.8 Dealing with time

It is quite usual in statecharts to find notations such as "*after 30 seconds*", often expressed as specific events on a transition. Sismic does not support the use of these *special events*, and proposes instead to deal with time by making use of some specifics provided by its interpreter and the default Python code evaluator.

Every interpreter has an internal clock whose value is initially set to 0. This internal clock is exposed by by the *time* property of an *Interpreter*. This property allows one to execute a statechart using simulated time. In other word, the value of this property won't change, unless you set it by yourself.

The built-in Python code evaluator allows one to make use of `after(...)`, `idle(...)` in guards or contracts. These two Boolean predicates can be used to automatically compare the current time (as exposed by the interpreter) with a predefined value that depends on the state in which the predicate is used. For instance, `after(x)` will evaluate to `True` if the current time of the interpreter is at least `x` units greater than the time when the state using this predicate (or source state in the case of a transition) was entered. Similarly, `idle(x)` evaluates to `True` if no transition was triggered during the last `x` units of time.

Note that while this property was initially designed to manage simulate time, it can also be used to synchronise the internal clock of an interpreter with the *real* time, i.e. wall-clock time.

2.8.1 Simulated time

The following example illustrates a statechart modeling the behavior of a simple *elevator*. If the elevator is sent to the 4th floor then, according to the YAML definition of this statechart, the elevator should automatically go back to the ground floor after 10 seconds.

```
- target: doorsClosed
  guard: after(10) and current > 0
  action: destination = 0
```

Rather than waiting for 10 seconds, one can simulate this. First, one should load the statechart and initialize the interpreter:

```
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter, Event

statechart = import_from_yaml(filepath='examples/elevator/elevator.yaml')

interpreter = Interpreter(statechart)
```

The internal clock of our interpreter is `0`. This is, `interpreter.time == 0` holds. We now ask our elevator to go to the 4th floor.

```
interpreter.queue(Event('floorSelected', floor=4))
interpreter.execute()
```

The elevator should now be on the 4th floor. We inform the interpreter that 2 seconds have elapsed:

```
interpreter.time += 2
print(interpreter.execute())
```

The output should be an empty list `[]`. Of course, nothing happened since the condition `after(10)` is not satisfied yet. We now inform the interpreter that 8 additional seconds have elapsed.

```
interpreter.time += 8
print(interpreter.execute())
```

The output now contains a list of steps, from which we can see that the elevator has moved down to the ground floor. We can check the current floor:

```
print(interpreter.context.get('current'))
```

This displays `0`.

2.8.2 Real or wall-clock time

If a statechart needs to be aware of a real clock, the simplest way to achieve this is by using the `time.time()` function of Python. In a nutshell, the idea is to synchronize `interpreter.time` with a real clock. Let us first initialize an interpreter using one of our statechart example, the *elevator*:

```
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter, Event

statechart = import_from_yaml(filepath='examples/elevator/elevator.yaml')

interpreter = Interpreter(statechart)
```

The interpreter initially sets its clock to 0. As we are interested in a real-time simulation of the statechart, we need to set the internal clock of our interpreter. We import from `time` a real clock, and store its value into a `starttime` variable.

```
import time
starttime = time.time()
```

We can now execute the statechart by sending a `floorSelected` event, and wait for the output. For our example, we first ask the statechart to send to elevator to the 4th floor.

```
interpreter.queue(Event('floorSelected', floor=4))
interpreter.execute()
print('Current floor:', interpreter.context.get('current'))
print('Current time:', interpreter.time)
```

At this point, the elevator is on the 4th floor and is waiting for another input event. The internal clock value is still 0.

```
Current floor: 4
Current time: 0
```

We should inform our interpreter of the new current time. Of course, as our interpreter follows a discrete simulation, nothing really happens until we call *`execute()`* or *`execute_once()`*.

```
interpreter.time = time.time() - starttime
# Does nothing if (time.time() - starttime) is less than 10!
interpreter.execute()
```

Assuming you quickly wrote these lines of code, nothing happened. But if you wait a little bit, and update the clock again, it should move the elevator to the ground floor.

```
interpreter.time = time.time() - starttime
interpreter.execute()
```

And *voilà*!

As it is not very convenient to manually set the clock each time you want to execute something, it is best to put it in a loop. To avoid the use of a `starttime` variable, you can set the initial time of an interpreter right after its initialization. This is illustrated in the following example.

```
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter, import Event

import time

# Load statechart and create an interpreter
statechart = import_from_yaml(filepath='examples/elevator.yaml')

# Set the initial time
interpreter = Interpreter(statechart)
interpreter.time = time.time()

# Send an initial event
interpreter.queue(Event('floorSelected', floor=4))

while not interpreter.final:
    interpreter.time = time.time()
    if interpreter.execute():
        print('something happened at time {}'.format(interpreter.time))

    time.sleep(0.5)  # 500ms
```

Here, we called the `sleep()` function to slow down the loop (optional). The output should look like:

```
something happened at time 1450383083.9943285
something happened at time 1450383093.9920669
```

As our statechart does not define any way to reach a final configuration, the `not interpreter.final` condition always holds, and the execution needs to be interrupted manually.

2.8.3 Asynchronous execution

Notice from previous example that using a loop makes it impossible to send events to the interpreter. For convenience, sismic provides a *`sismic.helpers.run_in_background()`* function that run an interpreter in a thread, and does the job of synchronizing the clock for you.

`sismic.helpers.`**`run_in_background`**(*interpreter*, *delay=0.05*, *callback=None*)

Run given interpreter in background. The time is updated according to *time.time() - starttime*. The interpreter is ran until it reaches a final configuration. You can manually stop the thread using the added *stop* of the returned Thread object. This is for convenience only and should be avoided, because a call to *stop* puts the interpreter in an empty (and thus final) configuration, without properly leaving the active states.

Parameters

- **`interpreter`** (`Interpreter`) – an interpreter
- **`delay`** (`float`) – delay between each call to *execute()*
- **`callback`** (`Optional[Callable[[List[MacroStep]], Any]]`) – a function that accepts the result of *execute*.

Return type `Thread`

Returns started thread (instance of *threading.Thread*)

Note: An optional argument `callback` can be passed to `run_in_background()`. It must be a callable that accepts the (possibly empty) list of `MacroStep` returned by the underlying call to `execute()`.

2.9 Integrate statecharts into your code

Sismic provides several ways to integrate executable statecharts into your Python source code. The simplest way is to directly *embed* the entire code in the statechart's description. This was illustrated with the Elevator example in *Include code in statecharts*. Its code is part of the YAML file of the statechart, and interpreted by Sismic during the statechart's execution.

In order to make a statechart communicate with the source code contained in the environment in which it is executed, there are basically two approaches:

1. The statechart sends events to, or receives external events from the environment.
2. The environment stores shared objects in the statechart's initial context, and the statechart calls operations on these objects and/or accesses the variables contained in it.

Of course, one could also use a hybrid approach, combining ideas from the three approches above.

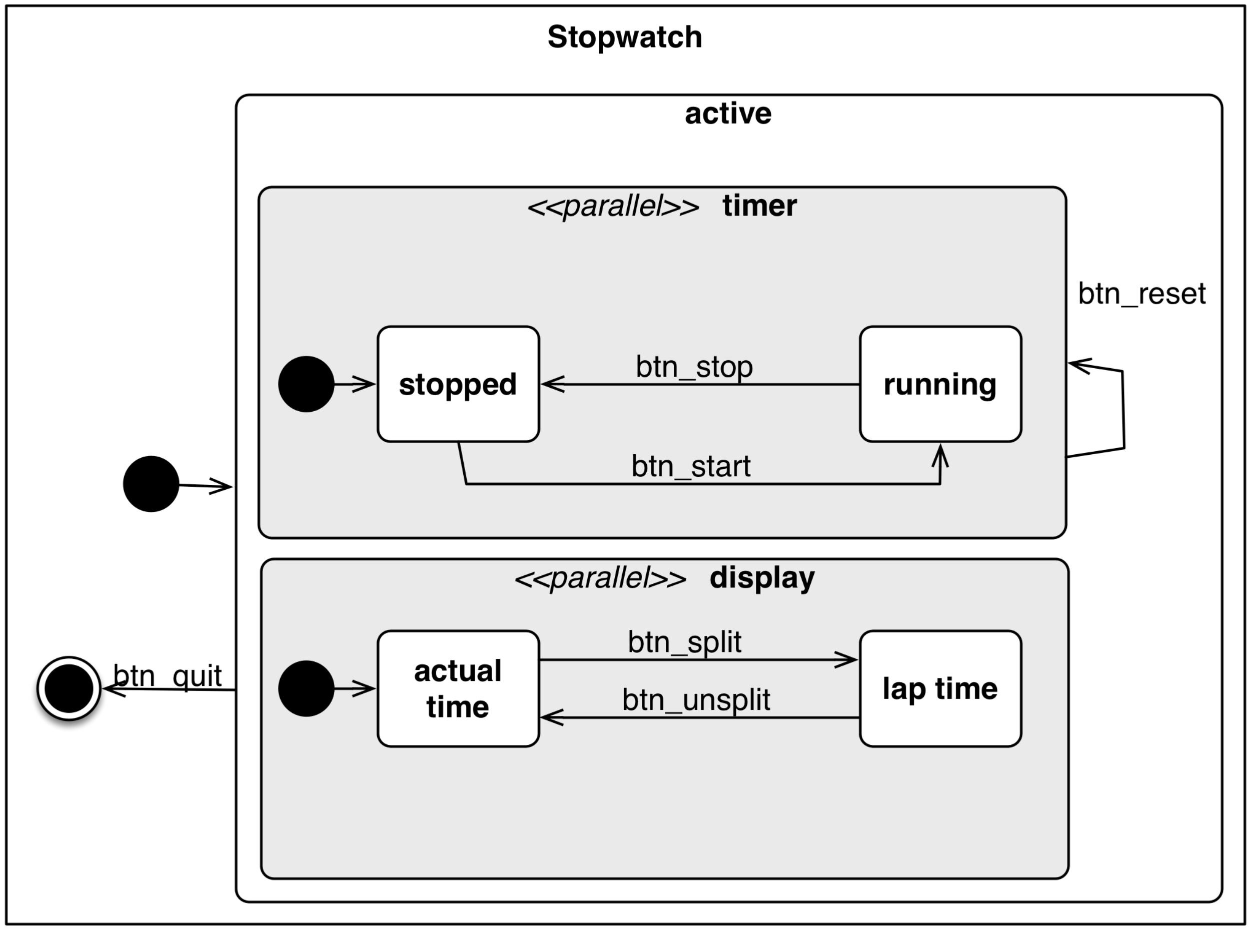
2.9.1 Running example

In this document, we will present the main differences between the two approaches, on the basis of a simple example of a Graphical User Interface (GUI) whose behaviour is defined by a statechart. All the source code and YAML files for this example, discussed in more detail below, is available in the *docs/examples* directory of Sismic's repository.

The example represents a simple stopwatch, i.e., a timer than can be started, stopped and reset. It also provides a split time feature and a display of the elapsed time. A button-controlled GUI of such a stopwatch looks as follows (inactive buttons are greyed out):

Essentially, the stopwatch simply displays a value, representing the elapsed time (expressed in seconds), which is initially 0. By clicking on the *start* button the stopwatch starts running. When clicking on *stop*, the stopwatch stops running. By using *split*, the time being displayed is temporarily frozen, although the stopwatch continues to run. Clicking on *unsplit* while continue to display the actual elapsed time. *reset* will restart from 0, and *quit* will quit the stopwatch application.

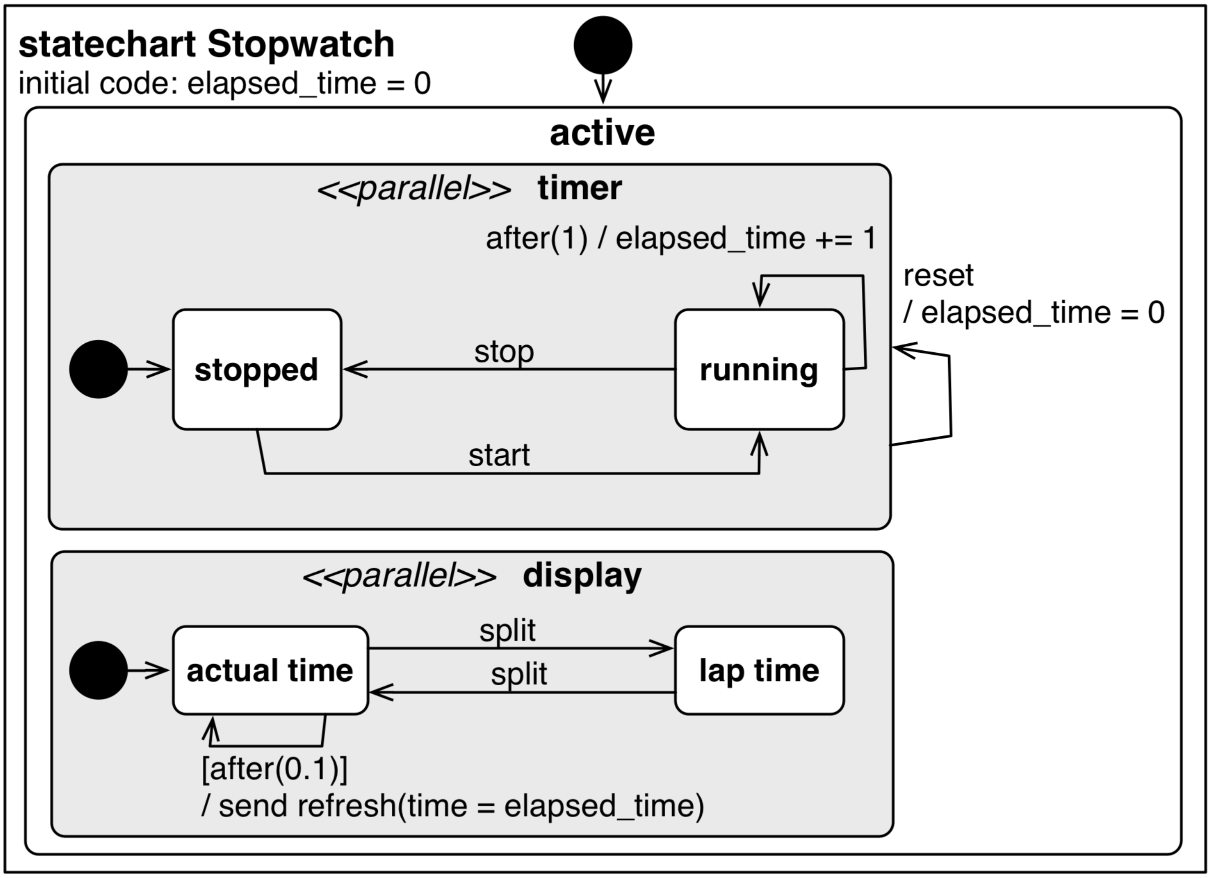
The idea is that the buttons will trigger state changes and actions carried out by an underlying statechart. Taking abstraction of the concrete implementation, the statechart would essentially look as follows, with one main *active* state containing two parallel substates *timer* and *display*.



2.9.2 Controlling a statechart from within the environment

Let us illustrate how to control a statechart through source code that executes in the environment containing the statechart. The statechart's behaviour is triggered by external events sent to it by the source code each time one of the buttons in the GUI is pressed. Conversely, the statechart itself can send events back to the source code to update its display.

This statechart looks as follows:



Here is the YAML file containing the textual description of this statechart:

```
statechart:
  name: Stopwatch
  description: |
    A simple stopwatch which support "start", "stop", "split", and "reset".
    These features are triggered respectively using "start", "stop", "split", and "reset".

    The stopwatch sends an "refresh" event each time the display is updated.
    The value to display is attached to the event under the key "time".

    The statechart is composed of two parallel regions:
     - A "timer" region which increments "elapsed_time" if timer is running
     - A "display" region that refreshes the display according to the actual time/lap time feature

  preamble: elapsed_time = 0
  root state:
    name: active
    parallel states:
      - name: timer
        initial: stopped
        transitions:
          - event: reset
            action: elapsed_time = 0
        states:
```

(continues on next page)

(continued from previous page)

```
          - name: running
            transitions:
              - event: stop
                target: stopped
              - guard: after(1)
                target: running
                action: elapsed_time += 1
          - name: stopped
            transitions:
              - event: start
                target: running
      - name: display
        initial: actual time
        states:
          - name: actual time
            transitions:
              - guard: after(0.2)
                target: actual time
                action: |
                  send('refresh', time=elapsed_time)
              - event: split
                target: lap time
          - name: lap time
            transitions:
              - event: split
                target: actual time
```

We observe that the statechart contains an `elapsed_time` variable, that is updated every second while the stopwatch is in the *running* state. The statechart will modify its behaviour by receiving *start*, *stop*, *reset* and *split* events from its external environment. In parallel to this, every 100 milliseconds, the *display* state of the statechart sends a *refresh* event (parameterised by the `time` variable containing the `elapsed_time` value) back to its external environment. In the *lap time* state (reached through a *split* event) , this regular refreshing is stopped until a new *split* event is received.

The source code (shown below) that defines the GUI of the stopwatch, and that controls the statechart by sending it events, is implemented using the `Tkinter` library. Each button of the GUI is bound to a Python method in which the corresponding event is created and sent to the statechart. The statechart is *bound* to the source code by defining a new *Interpreter* that contains the parsed YAML specification, and using the *bind()* method. The `event_handler` passed to it allows the Python source code to receive events back from the statechart. In particular, the `w_timer` field of the GUI will be updated with a new value of the time whenever the statechart sends a *refresh* event. The `run` method, which is put in Tk's mainloop, updates the internal clock of the interpreter and executes the interpreter.

```
import time
import tkinter as tk

from sismic.interpreter import Interpreter, Event
from sismic.io import import_from_yaml


# The two following lines are NOT needed in a typical environment.
# These lines make sismic available in our testing environment
import sys
sys.path.append('../../..')
```

(continues on next page)

(continued from previous page)

```
# Create a tiny GUI
class StopwatchApplication(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)

        # Initialize widgets
        self.create_widgets()

        # Create a Stopwatch interpreter
        with open('stopwatch.yaml') as f:
            statechart = import_from_yaml(f)
        self.interpreter = Interpreter(statechart)
        self.interpreter.time = time.time()

        # Bind interpreter events to the GUI
        self.interpreter.bind(self.event_handler)

        # Run the interpreter
        self.run()

    def run(self):
        # This function does essentially the same job than ``sismic.interpreter.run_
↪in_background``
        # but uses Tkinter's mainloop instead of a Thread, which is more adequate.

        # Update internal clock and execute interpreter
        self.interpreter.time = time.time()
        self.interpreter.execute()

        # Queue a call every 100ms on tk's mainloop
        self.after(100, self.run)

        # Update the widget that contains the list of active states.
        self.w_states['text'] = 'active states: ' + ', '.join(self.interpreter.
↪configuration)

    def create_widgets(self):
        self.pack()

        # Add buttons
        self.w_btn_start = tk.Button(self, text='start', command=self._start)
        self.w_btn_stop = tk.Button(self, text='stop', command=self._stop)
        self.w_btn_split = tk.Button(self, text='split', command=self._split)
        self.w_btn_unsplit = tk.Button(self, text='unsplit', command=self._unsplit)
        self.w_btn_reset = tk.Button(self, text='reset', command=self._reset)
        self.w_btn_quit = tk.Button(self, text='quit', command=self._quit)

        # Initial button states
        self.w_btn_stop['state'] = tk.DISABLED
        self.w_btn_unsplit['state'] = tk.DISABLED

        # Pack
        self.w_btn_start.pack(side=tk.LEFT,)
        self.w_btn_stop.pack(side=tk.LEFT,)
        self.w_btn_split.pack(side=tk.LEFT,)
        self.w_btn_unsplit.pack(side=tk.LEFT,)
```

(continues on next page)

(continued from previous page)

```
        self.w_btn_reset.pack(side=tk.LEFT,)
        self.w_btn_quit.pack(side=tk.LEFT,)

        # Active states label
        self.w_states = tk.Label(root)
        self.w_states.pack(side=tk.BOTTOM, fill=tk.X)

        # Timer label
        self.w_timer = tk.Label(root, font=("Helvetica", 16), pady=5)
        self.w_timer.pack(side=tk.BOTTOM, fill=tk.X)

    def event_handler(self, event):
        # Update text widget when timer value is updated
        if event.name == 'refresh':
            self.w_timer['text'] = event.time

    def _start(self):
        self.interpreter.queue(Event('start'))
        self.w_btn_start['state'] = tk.DISABLED
        self.w_btn_stop['state'] = tk.NORMAL

    def _stop(self):
        self.interpreter.queue(Event('stop'))
        self.w_btn_start['state'] = tk.NORMAL
        self.w_btn_stop['state'] = tk.DISABLED

    def _reset(self):
        self.interpreter.queue(Event('reset'))

    def _split(self):
        self.interpreter.queue(Event('split'))
        self.w_btn_split['state'] = tk.DISABLED
        self.w_btn_unsplit['state'] = tk.NORMAL

    def _unsplit(self):
        self.interpreter.queue(Event('split'))
        self.w_btn_split['state'] = tk.NORMAL
        self.w_btn_unsplit['state'] = tk.DISABLED

    def _quit(self):
        self.master.destroy()


if __name__ == '__main__':
    # Create GUI
    root = tk.Tk()
    root.wm_title('StopWatch')
    app = StopwatchApplication(master=root)

    app.mainloop()
```

2.9.3 Controlling the environment from within the statechart

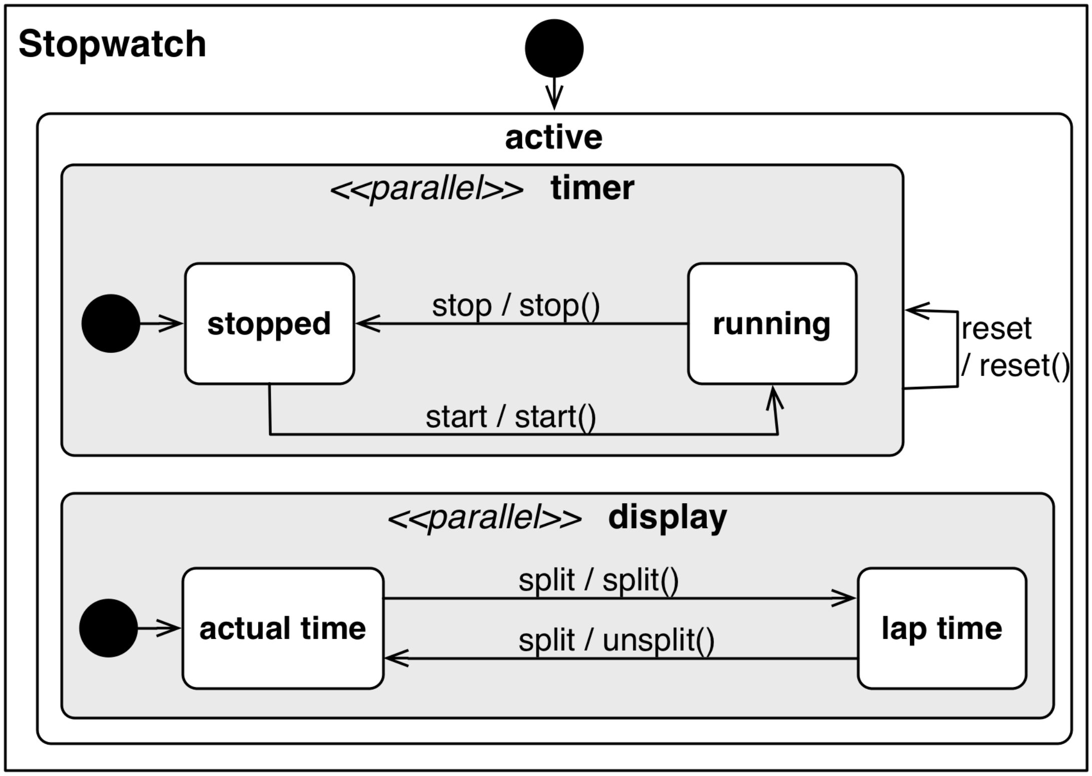
In this second example, we basically reverse the idea: now the Python code that resides in the environment contains the logic (e.g., the `elapsed_time` variable), and this code is exposed to, and controlled by, a statechart that represents the main loop of the program and calls the necessary methods in the source code. These method calls are associated to actions on the statechart's transitions. With this solution, the statechart is no longer a *black box*, since it needs to be aware of the source code, in particular the methods it needs to call in this code.

An example of the Python code that is controlled by the statechart is given below:

```
class Stopwatch:
    def __init__(self):
        self.elapsed_time = 0
        self.split_time = 0
        self.is_split = False
        self.running = False

    def start(self):
        # Start internal timer
        self.running = True

    def stop(self):
        # Stop internal timer
        self.running = False

    def reset(self):
        # Reset internal timer
        self.elapsed_time = 0

    def split(self):
        # Split time
        if not self.is_split:
            self.is_split = True
            self.split_time = self.elapsed_time

    def unsplit(self):
        # Unsplit time
        if self.is_split:
            self.is_split = False

    def display(self):
        # Return the value to display
        if self.is_split:
            return int(self.split_time)
        else:
            return int(self.elapsed_time)

    def update(self, delta):
        # Update internal timer of ``delta`` seconds
        if self.running:
            self.elapsed_time += delta
```

The statechart expects such a `Stopwatch` instance to be created and provided in its initial context. Recall that an *`Interpreter`* accepts an optional `initial_context` parameter. In this example, `initial_context={'stopwatch': Stopwatch()}`.

The statechart is simpler than in the previous example: one parallel region handles the running status of the stopwatch, and a second one handles its split features.




```
statechart:
  name: Stopwatch
  description: |
    A simple stopwatch which support "start", "stop", "split", and "reset".
    These features are triggered respectively using "start", "stop", "split", and
↪"reset".

    The stopwatch expects a "stopwatch" object in its initial context.
    This object should support the following methods: "start", "stop", "split", "reset
↪", and "unsplit".
  root state:
    name: active
    parallel states:
      - name: timer
        initial: stopped
        transitions:
          - event: reset
            action: stopwatch.reset()
        states:
          - name: running
            transitions:
              - event: stop
                target: stopped
                action: stopwatch.stop()
          - name: stopped
            transitions:
              - event: start
                target: running
                action: stopwatch.start()
      - name: display
```

(continues on next page)

(continued from previous page)

```
        initial: actual time
        states:
          - name: actual time
            transitions:
              - event: split
                target: lap time
                action: stopwatch.split()
          - name: lap time
            transitions:
              - event: split
                target: actual time
                action: stopwatch.unsplit()
```

The Python code of the GUI no longer needs to *listen* to the events sent by the interpreter. It should, of course, continue to send events (corresponding to button presses) to the statechart using `send`. The *binding* between the statechart and the GUI is now achieved differently, by simply passing the `stopwatch` object to the *Interpreter* as its `initial_context`.

```
import time
import tkinter as tk

from sismic.interpreter import Interpreter, Event
from sismic.io import import_from_yaml
from stopwatch import Stopwatch


# The two following lines are NOT needed in a typical environment.
# These lines make sismic available in our testing environment
import sys
sys.path.append('../../..')




# Create a tiny GUI
class StopwatchApplication(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)

        # Initialize widgets
        self.create_widgets()

        # Create a Stopwatch interpreter
        with open('stopwatch_external.yaml') as f:
            statechart = import_from_yaml(f)

        # Create a stopwatch object and pass it to the interpreter
        self.stopwatch = Stopwatch()
        self.interpreter = Interpreter(statechart, initial_context={'stopwatch': self.
↪stopwatch})
        self.interpreter.time = time.time()

        # Run the interpreter
        self.run()

        # Update the stopwatch every 100ms
```

(continues on next page)

(continued from previous page)

```
        self.after(100, self.update_stopwatch)

    def update_stopwatch(self):
        self.stopwatch.update(delta=0.1)
        self.after(100, self.update_stopwatch)

        # Update timer label
        self.w_timer['text'] = self.stopwatch.display()

    def run(self):
        # Queue a call every 100ms on tk's mainloop
        self.interpreter.execute()
        self.after(100, self.run)
        self.w_states['text'] = 'active states: ' + ', '.join(self.interpreter.
↪configuration)

    def create_widgets(self):
        self.pack()

        # Add buttons
        self.w_btn_start = tk.Button(self, text='start', command=self._start)
        self.w_btn_stop = tk.Button(self, text='stop', command=self._stop)
        self.w_btn_split = tk.Button(self, text='split', command=self._split)
        self.w_btn_unsplit = tk.Button(self, text='unsplit', command=self._unsplit)
        self.w_btn_reset = tk.Button(self, text='reset', command=self._reset)
        self.w_btn_quit = tk.Button(self, text='quit', command=self._quit)

        # Initial button states
        self.w_btn_stop['state'] = tk.DISABLED
        self.w_btn_unsplit['state'] = tk.DISABLED

        # Pack
        self.w_btn_start.pack(side=tk.LEFT,)
        self.w_btn_stop.pack(side=tk.LEFT,)
        self.w_btn_split.pack(side=tk.LEFT,)
        self.w_btn_unsplit.pack(side=tk.LEFT,)
        self.w_btn_reset.pack(side=tk.LEFT,)
        self.w_btn_quit.pack(side=tk.LEFT,)

        # Active states label
        self.w_states = tk.Label(root)
        self.w_states.pack(side=tk.BOTTOM, fill=tk.X)

        # Timer label
        self.w_timer = tk.Label(root, font=("Helvetica", 16), pady=5)
        self.w_timer.pack(side=tk.BOTTOM, fill=tk.X)

    def _start(self):
        self.interpreter.queue(Event('start'))
        self.w_btn_start['state'] = tk.DISABLED
        self.w_btn_stop['state'] = tk.NORMAL

    def _stop(self):
        self.interpreter.queue(Event('stop'))
        self.w_btn_start['state'] = tk.NORMAL
        self.w_btn_stop['state'] = tk.DISABLED
```

(continues on next page)

(continued from previous page)

```
    def _reset(self):
        self.interpreter.queue(Event('reset'))

    def _split(self):
        self.interpreter.queue(Event('split'))
        self.w_btn_split['state'] = tk.DISABLED
        self.w_btn_unsplit['state'] = tk.NORMAL

    def _unsplit(self):
        self.interpreter.queue(Event('split'))
        self.w_btn_split['state'] = tk.NORMAL
        self.w_btn_unsplit['state'] = tk.DISABLED

    def _quit(self):
        self.master.destroy()


if __name__ == '__main__':
    # Create GUI
    root = tk.Tk()
    root.wm_title('StopWatch (external)')
    app = StopwatchApplication(master=root)

    app.mainloop()
```

2.10 Communication between statecharts

It is not unusual to have to deal with multiple distinct components in which the behavior of a component is driven by things that happen in the other components. One can model such a situation using a single statechart with parallel states, or by plugging several statecharts into one main statechart (see `sismic.model.Statechart.copy_from_statechart()`). The communication and synchronization between the components can be done either by using `active(state_name)` in guards, or by sending internal events that address other components.

However, we believe that this approach is not very convenient:

- all the components must be defined in a single statechart;
- state name collision could occur;
- components must share a single execution context;
- component composition is not easy to achieve
- …

Sismic allows to define multiple components in multiple statecharts, and brings a way for those statecharts to communicate and synchronize via events.

2.10.1 Binding statecharts

Every instance of `Interpreter` exposes a `bind()` method which allows to bind statecharts.

`Interpreter.bind(interpreter_or_callable)`

Bind an interpreter or a callable to the current interpreter. Each time an internal event is sent by this interpreter, any bound object will be called with the same event. If *interpreter_or_callable* is an *Interpreter* instance, its *queue* method is called. This is, if *i1* and *i2* are interpreters, *i1.bind(i2)* is equivalent to *i1.bind(i2.queue)*.

Parameters **interpreter_or_callable** (Union[Interpreter, Callable[[Event], Any]]) – interpreter or callable to bind

Return type None

Returns *self* so it can be chained

When an interpreter `interpreter_1` is bound to an interpreter `interpreter_2` using `interpreter_1.bind(interpreter_2)`, the **internal** events that are sent by `interpreter_1` are automatically propagated as **external** events to `interpreter_2`. The binding is not restricted to only two statecharts. For example, assume we have three instances of *Interpreter*:

```
assert isinstance(interpreter_1, Interpreter)
assert isinstance(interpreter_2, Interpreter)
assert isinstance(interpreter_3, Interpreter)
```

We define a bidirectional communication between the two first interpreters:

```
interpreter_1.bind(interpreter_2)
interpreter_2.bind(interpreter_1)
```

We also bind the third interpreters with the two first ones.

```
interpreter_3.bind(interpreter_1)
interpreter_3.bind(interpreter_2)
```

When an internal event is sent by an interpreter, the bound interpreters also receive this event as an external event. In the last example, when an internal event is sent by `interpreter_3`, then a corresponding external event is sent both to `interpreter_1` and `interpreter_2`.

Note: Practically, unless you subclassed *Interpreter*, the only difference between internal and external events are the priority order in which they are processed by the interpreter.

```
from sismic.interpreter import InternalEvent, Event

# Manually create and raise an internal event
interpreter_3._raise_event(InternalEvent('test'))

print('Events for interpreter_1:', interpreter_1._external_events.pop())
print('Events for interpreter_2:', interpreter_2._external_events.pop())
print('Events for interpreter_3:', interpreter_3._internal_events.pop())
```

```
Events for interpreter_1: Event('test')
Events for interpreter_2: Event('test')
Events for interpreter_3: InternalEvent('test')
```

2.10.2 Example of communicating statecharts

Consider our running example, the elevator statechart. This statechart expects to receive *floorSelected* events (with a *floor* parameter representing the selected floor). The statechart operates autonomously, provided that we send such events.

Let us define a new statechart that models a panel of buttons for our elevator. For example, we consider that our panel has 4 buttons numbered 0 to 3.

```yaml
statechart:
  name: Elevator buttons
  description: |
    Buttons that remotely control the elevator.
  root state:
    name: active
    parallel states:
      - name: button_0
        transitions:
          - event: button_0_pushed
            action: send('floorSelected', floor= 0)
      - name: button_1
        transitions:
          - event: button_1_pushed
            action: send('floorSelected', floor= 1)
      - name: button_2
        transitions:
          - event: button_2_pushed
            action: send('floorSelected', floor= 2)
      - name: button_3
        transitions:
          - event: button_3_pushed
            action: send('floorSelected', floor= 3)
```

As you can see in the YAML version of this statechart, the panel expects an event for each button: *button_0_pushed*, *button_1_pushed*, *button_2_pushed* and *button_3_pushed*. Each of those event causes the execution of a transition which, in turn, creates and sends a *floorSelected* event. The *floor* parameter of this event corresponds to the button number.

We bind our panel with our elevator, such that the panel can control the elevator:

```python
from sismic.io import import_from_yaml
from sismic.interpreter import Interpreter, Event, InternalEvent

elevator = Interpreter(import_from_yaml(filepath='examples/elevator/elevator.yaml'))
buttons = Interpreter(import_from_yaml(filepath='examples/elevator/elevator_buttons.
↪yaml'))

# Elevator will receive events from buttons
buttons.bind(elevator)
```

Events that are sent **to** `buttons` are not propagated, but events that are sent **by** `buttons` are automatically propagated to `elevator`:

```python
print('Awaiting events in buttons:', list(buttons._external_events))  # Empty
buttons.queue(Event('button_2_pushed'))

print('Awaiting events in buttons:', list(buttons._external_events))  # External event

buttons.execute(max_steps=2)  # (1) initialize buttons, and (2) consume button_2_
↪pushed
print('Awaiting events in buttons:', list(buttons._internal_events))
print('Awaiting events in elevator:', list(elevator._external_events))
```

```
Awaiting events in buttons: []
Awaiting events in buttons: [Event('button_2_pushed')]
Awaiting events in buttons: [InternalEvent('floorSelected', floor=2)]
```

(continues on next page)

(continued from previous page)

```
Awaiting events in elevator: [Event('floorSelected', floor=2)]
```

The execution of bound statecharts does not differ from the execution of unbound statecharts:

```
elevator.execute()
print('Current floor:', elevator.context.get('current'))
```

```
Current floor: 2
```

2.11 Extensions for Sismic

Sismic can be quite easily extended to support other semantics, other code evaluators or even other features. The sismic-extensions repository already provides some extensions. Feel free to contact us if you developed an extension you would want to be listed here.

2.11.1 sismic-amola

This extension provides support to import and export statechart written using AMOLA. This allows statecharts to be created, edited and displayed with the ASEME IDE. It exposes `import_from_amola` and `export_to_amola` based on the bundled Ecore meta-model (see *amola.ecore*).

Download: https://github.com/AlexandreDecan/sismic-extensions/tree/master/sismic_amola

2.11.2 sismic-semantics

This extension contains two variations around the default interpreter: one supporting outer-first/source-state semantics, and a second giving priority to transitions with event (instead of eventless transitions).

The extension provides two new interpreter classes: `OuterFirstInterpreter` and `EventFirstInterpreter`. These two interpreters can be combined together, thanks to Python multiple inheritance.

Download: https://github.com/AlexandreDecan/sismic-extensions/tree/master/sismic_semantics

2.12 Credits

2.12.1 Development Lead

- Alexandre Decan

2.12.2 Contributors

- Tom Mens
- Mathieu Goeminne
- Ali Parsai
- Nikos Spanoudakis

2.13 Changelog

2.13.1 1.1.2 (2018-05-09)

- (Fixed) Interpreter instances can be serialized using `pickle` (#66).

2.13.2 1.1.1 (2018-04-26)

- (Fixed) Whitespaces in event parameters used in BDD steps are stripped before they are evaluated.

2.13.3 1.1.0 (2018-04-23)

- (Fixed) Final states remain in the active configuration unless they are all children of the root state. In this case, statechart execution is stopped. Previously, if all leaf states of the active configuration were final states, the execution stopped even if these final states were nested in an orthogonal or compound state. The corrected behavior strictly adheres to SCXML 1.0 semantics. This could be a backward incompatible change if you explicitly relied on the previously wrong behaviour.
- (Added) `Interpreter._select_event` accepts an additional parameter `consume` (default to True) that can be used to select an event without consuming it.
- (Added) Documentation for extensions, and two (not included in Sismic!) extensions providing import/export with AMOLA, and new semantics for the interpreter.

2.13.4 1.0.1 (2018-04-18)

- (Fixed) BDD steps that involve a state raise a `StatechartError` if state does not exist. This prevents *state X is active* (and its variants) to fail, e.g., because *X* is misspelled.

2.13.5 1.0.0 (2018-04-11)

After more than two years of development, Sismic is stable enough to be released in version 1.0.0. Consequently, Sismic will adhere to semantic versioning (see semver.org), meaning that breaking changes will only occur in major releases, backward compatible changes in minor releases, and bug fixes in patches.

2.13.6 0.26.9 (2018-04-03)

- (Fixed) `based_on` for `export_to_plantuml` correctly takes into account states whose name contains whitespaces.
- (Fixed) `export_to_plantuml` properly exports transition with no event, no guard and no action.
- (Changed) `export_to_yaml` does not add quotes by default.

2.13.7 0.26.8 (2018-03-23)

- (Added) `import_from_yaml` accepts a `filepath` argument.

- (Added) `based_on` and `based_on_filepath` parameters for `export_to_plantuml` so a previously generated PlantUML file can be used as a basis for a new one (including its modifications related to the direction and length of transitions).

2.13.8 0.26.7 (2018-03-21)

- (Removed) Nested context (ie. nested variable scopes) for the Python code evaluator.
- (Fixed) BDD step *expression {expression} holds*.

2.13.9 0.26.6 (2018-03-17)

- (Changed) Export to PlantUML uses short arrows by default.
- (Changed) Many improvements related to the transitions when using `export_to_plantuml`.

2.13.10 0.26.4 (2018-03-16)

- (Added) `sismic.bdd.execute_bdd` can be used to execute BDD tests programmatically.
- (Added) `sismic.bdd.__main__` is the CLI interface for `sismic-behave` and can now be executed using `python -m sismic.bdd` too if sismic is available but not installed.
- (Added) Many tests for BDD steps.
- (Changed) `Statechart.copy_from_statechart` has only its first argument that can be provided by position. The remaining ones (esp. `source` and `replace`) should be provided by name.
- (Fixed) Sismic requires behave >= 1.6.0.
- (Fixed) Older versions of typing do not contain `Deque`.
- (Removed) `sismic.bdd.cli.execute_behave`, subsumed by `sismic.bdd.execute_bdd`.

2.13.11 0.26.3 (2018-03-15)

- (Added) `sismic.bdd` exposes `sismic.bdd.cli.execute_behave` function to programmatically use `sismic-bdd`.
- (Changed) `execute_behave` function has only two required parameters, and the remaining ones (that have default values) can only be set by name, not by position.
- (Changed) `action_alias` and `assertion_alias` of module `sismic.bdd.steps` are renamed to `map_action` and `map_assertion` and are directly available from `sismic.bdd`.

2.13.12 0.26.2 (2018-03-15)

- (Fixed) Step *Given/when I repeat "{step}" {repeat} times* requires *step* to be provided with no Gherkin keyword. The current keyword (either *given* or *when*) is automatically used.
- (Fixed) Escape expression in *then expression "{expression}" holds* and its negative counterpart.

2.13.13 0.26.0 (2018-03-15)

Sismic support for BDD was completely rewritten. The CLI is now `sismic-bdd`, pointing to the `cli` submodule of the newly created `sismic.bdd` module. All steps that are related to Sismic internals were removed, and only steps that manipulate the statechart are kept. Check the documentation and `sismic.bdd.steps` for more information. Execution semantics have slightly changed but shouldn't have any impact when running BDD tests. Predefined steps can be easily extended thanks to the `action_alias` and `assertion_alias` helpers. See documentation for more details.

- (Changed) `sismic-behave` CLI is now `sismic-bdd`.
- (Removed) `--coverage` option from `sismic-behave` CLI.
- (Changed) Rename `sismic.testing` to `sismic.bdd`, and `sismic.testing.behave` to `sismic.bdd.cli`.
- (Changed) A new list of predefined steps, available in `sismic.bdd.steps`, see documentation.
- (Changed) A "when" step is now required before any "then" step. The "then" steps assert on what happens during the "when" steps, and not on the whole execution or the last step as before.
- (Added) `sismic.bdd.steps` provides `action_alias` and `assertion_alias` to make defining new steps easy.
- (Changed) BDD tests are directly executed by `pytest` (instead of being triggered by Travis-CI).

Other changes:

- (Changed) `Interpreter.bind_property` becomes `Interpreter.bind_property_statechart`.
- (Changed) `helpers.coverage_from_trace` returns a dict with "entered states", "exited states" and "processed transitions".
- (Removed) Unused `io.text`.

2.13.14 0.25.3 (2018-03-13)

- (Fixed) `export_to_dict` (and by extension, `export_to_yaml`) didn't export transition contracts.
- (Changed) All the tests are now written using `pytest` instead of `unittest`.

2.13.15 0.25.2 (2018-03-11)

- (Added) Make `Event`, `InternalEvent` and `MetaEvent` available from `interpreter` as well.
- (Changed) Move `helpers` from `sismic.interpreter.helpers` to `sismic.helpers`.
- (Removed) Remove module `stories`, not really required anymore.

2.13.16 0.25.1 (2018-03-09)

- (Added) Full equality comparison (`__eq__`) for states and transitions (including all relevant attributes).
- (Added) `Interpreter.queue` also accepts an event name in addition to an `Event` instance.
- (Added) `Interpreter.queue` accepts more than one event (or name) at once.
- (Changed) `Evaluator.execute_onentry` and `execute_onexit` become `execute_on_entry` and `execute_on_exit`.

- (Changed) Many type annotations were added or fixed.
- (Changed) `Interpreter.bind` can no longer be chained.

2.13.17 0.25.0 (2018-03-09)

Property statecharts do not require anymore the use of an `ExecutionWatcher` and are now directly supported by the interpreter. The documentation contains a new page, *Monitoring properties*, that explains how to monitor properties at runtime and provides some examples of property statecharts.

- (Added) Property statechart can be bound to an interpreter with `interpreter.bound_property` method, that accepts either a `Statechart` or an `Interpreter` instance.
- (Added) A `PropertyStatechartError` that is raised when a property statechart reaches a final state.
- (Added) A `MetaEvent` class to represent meta-events sent by the interpreter for property statechart checking.
- (Added) `Interpreter._notify_property(event_name, **kwargs)` and `Interpreter._check_properties(macro_step)` that are used internally to respectively send meta-events to bound properties, and to check these properties.
- (Changed) `Interpreter.raise_event` is now `Interpreter._raise_event` as it's not supposed to be part of the public API.
- (Removed) `sismic.testing` module was removed (including the `ExecutionWatcher` and `TestStoryFromTrace`).
- (Removed) BDD steps related to the execution watcher, in `sismic.testing.steps`.
- (Fixed) `Interpreter.time` cannot be set to a lower value than the current one (ie. time is monotonic).
- (Fixed) A statechart preamble cannot be used to send events.

2.13.18 0.24.3 (2018-03-08)

- (Fixed) `ExecutionWatcher.stop()` was not called at the end of the execution when `sismic-behave` was called with `--properties`.
- (Removed) Unused dependency on `pyparsing`.

2.13.19 0.24.2 (2018-02-27)

- (Added) `sismic.io` contains an `export_to_plantuml` function to export a statechart to PlantUML.
- (Added) `sismic-behave` accepts a `--properties` argument, pointing to a list of YAML files containing property statecharts that will be checked during execution (in a fail fast mode).
- (Changed) `sismic.io.export_to_yaml` accepts an additional `filepath` argument.
- (Fixed) Whitespaces in strings are trimmed when using `import_from_dict` (and hence, using `import_from_yaml`).

2.13.20 0.23.1 (2018-02-20)

- (Fixed) An exited state is removed from the current configuration before its postconditions are checked.
- (Removed) Sequential conditions that were introduced in 0.22.0.

2.13.21 0.22.11 (2017-01-12)

- (Fixed) Path error when using `sismic-behave` on Windows.

2.13.22 0.22.10 (2016-11-25)

- (Added) A `--debug-on-error` parameter for `sismic-behave`.

2.13.23 0.22.9 (2016-11-25)

- (Fixed) Behave step "Event x should be fired" now checks that the event was fired during the last execution.

2.13.24 0.22.8 (2016-10-19)

- (Fixed) YAML values like "1", "1.0", "yes", "True" are converted to strings, not to int, float and bool respectively.
- (Changed) `ruamel.yaml` replaces `pyyaml` as supported YAML parser.
- (Changed) Use `schema` instead of `pykwalify` (which unfortunately freezes its dependencies versions) to validate (the structure of) YAML files.
- (Changed) `import_from_yaml` raises `StatechartError` instead of `SchemaError` if it cannot validate given YAML against the predefined schema.

2.13.25 0.22.7 (2016-08-19)

- (Added) A new helper `coverage_from_trace` that returns coverage information (in absolute numbers) from a trace.
- (Added) Parameter `fails_fast` (default is `False`, behavior preserved) for `ExecutionWatcher.watch_with` methods. This parameter allows the watcher to raise an `AssertionError` as soon as the added watcher reaches a final configuration.
- (Changed) `StateMixin`, `Transition` and `Event`'s `__eq__` method returns a `NotImplemented` object if the other object involved in the comparison is not an instance of the same class, meaning that `Event('a') == 1` now raises a `NotImplementedError` instead of being `False`.

2.13.26 0.22.6 (2016-08-03)

- (Changed) `Event`, `MacroStep`, `MicroStep`, `StateMixin`, `Transition`, `Statechart` and `Interpreter`'s `__repr__` returns a valid Python expression.
- (Changed) The context returned by a `PythonEvaluator` (and thus by the default `Interpreter`) exhibits nested variables (the ones that are not defined in the preamble of a statechart). Those variables are prefixed by the name of the state in which they are declared, to avoid name clashing.
- (Changed) Context variables are sorted in exceptions' `.__str__` methods.

2.13.27 0.22.4 (2016-07-08)

- (Added) `sismic-behave` CLI now accepts a `--steps` parameter, which is a list of file paths containing the steps implementation.
- (Added) `sismic-behave` CLI now accepts a `--show-steps` parameter, which list the steps (equivalent to Behave's overriden `--steps` parameter).
- (Added) `sismic-behave` now returns an appropriate exit code.
- (Changed) Reorganisation of `docs/examples`.
- (Fixed) Coverage data for `sismic-behave` takes the initialization step into account (regression introduced in 0.21.0).

2.13.28 0.22.3 (2016-07-06)

- (Added) `sent` and `received` are also available in preconditions and postconditions.

2.13.29 0.22.2 (2016-07-01)

- (Added) `model.Event` is now correctly pickled, meaning that Sismic can be used in a multiprocessing environment.

2.13.30 0.22.1 (2016-06-29)

- (Added) A *event {event_name} should not be fired* steps for BDD.
- (Added) Both `MicroStep` and `MacroStep` have a list `sent_events` of events that were sent during the step.
- (Added) Property statecharts receive a `event sent` event when an event is sent by the statechart under test.
- (Changed) Events fired from within the statechart are now collected and sent at the end of the current micro step, instead of being immediately sent.
- (Changed) Invariants and sequential contracts are now evaluated ordered by their state's depth

2.13.31 0.22.0 (2016-06-13)

- (Added) Support for sequential conditions in contracts (see documentation for more information).
- (Added) Python code evaluator: *after* and *idle* are now available in postconditions and invariants.
- (Added) Python code evaluator: *received* and *sent* are available in invariants.
- (Added) An `Evaluator` has now a `on_step_starts` method which is called at the beginning of each step, with the current event (if any) being processed.
- (Added) `Interpreter.raise_event` to send events from within the statechart.
- (Added) A `copy_from_statechart` method for a `Statechart` instance that allows to copy (part of) a statechart into a state.
- (Added) Microwave controller example (see *docs/examples/microwave.[yaml|py]*).
- (Changed) Events sent by a code evaluator are now returned by the `execute_*` methods instead of being automatically added to the interpreter's queue.

- (Changed) Moved `run_in_background` and `log_trace` from `sismic.interpreter` to the newly added `sismic.interpreter.helpers`.
- (Changed) Internal API changes: rename `self.__x` to `self._x` to avoid (mostly) useless name mangling.

2.13.32 0.21.0 (2016-04-22)

Changes for `interpreter.Interpreter` class:

- (Removed) `_select_eventless_transition` which is a special case of `_select_transition`.
- (Added) `_select_event`, extracted from `execute_once`.
- (Added) `_filter_transitions`, extracted from `_select_transition`.
- (Changed) `_execute_step` is now `_apply_step`.
- (Changed) `_compute_stabilization_step` is now `_create_stabilization_step` and accepts a list of state names
- (Changed) `_compute_transitions_step` is now `_create_steps`.
- (Changed) Except for the `statechart` parameter, all the parameters for `Interpreter`'s constructor can now be only provided by name.
- (Fixed) Contracts on a transition are checked (if not explicitly disabled) even if the transition has no *action*.
- (Fixed) `Evaluator.execute_action` is called even if the transition has no *action*.
- (Fixed) States are added/removed from the active configuration as soon as they are entered/exited. Previously, the configuration was only updated at the end of the step (and could possibly lead to inaccurate results when using `active(name)` in a `PythonEvaluator`).

The default `PythonEvaluator` class has been completely rewritten:

- (Changed) Code contained in states and/or transitions is now executed with a local context instead of a global one. The local context of a state is built upon the local context of its parent, and so one until the local context of the statechart is reached. This should facilitate the use of dummy variables in nested states and transitions.
- (Changed) The code is now compiled (once) before is evaluation/execution. This should increase performance.
- (Changed) The frozen context of a state (ie. `__old__`) is now computed only if contracts are checked, and only if at least one invariant or one postcondition exists.
- (Changed) The `initial_context` parameter of `Evaluator`'s constructor can now only be provided by name.
- (Changed) The `additional_context` parameter of `Evaluator._evaluate_code` and `Evaluator._execute_code` can now only be provided by name.

Miscellaneous:

- (Fixed) Step *I load the statechart* now executes (once) the statechart in order to put it into a stable initial configuration (regression introduced in 0.20.0).

2.13.33 0.20.5 (2016-04-14)

- (Added) Type hinting (see PEP484 and mypy-lang project)

2.13.34 0.20.4 (2016-03-25)

- (Changed) Statechart testers are now called property statechart.
- (Changed) Property statechart can describe *desirable* and *undesirable* properties.

2.13.35 0.20.3 (2016-03-22)

- (Changed) Step *Event x should be fired* now checks sent events from the beginning of the test, not only for the last executed step.
- (Fixed) Internal events that are sequentially sent are now sequentially consumed (and not anymore in reverse order).

2.13.36 0.20.2 (2016-02-24)

- (Fixed) `interpreter.log_trace` does not anymore log empty macro step.

2.13.37 0.20.1 (2016-02-19)

- (Added) A *step ended* event at the end of each step in a tester story.
- (Changed) The name of the events and attributes that are exposed in a tester story has changed. Consult the documentation for more information.

2.13.38 0.20.0 (2016-02-17)

- (Added) Module `interpreter` provides a `log_trace` function that takes an interpreter instance and returns a (dynamic) list of executed macro steps.
- (Added) Module `testing` exposes an `ExecutionWatcher` class that can be used to check statechart properties with tester statecharts at runtime.
- (Changed) `Interpreter.__init__` does not anymore stabilize the statechart. Stabilization is done during the first call of `execute_once`.
- (Changed) `Story.tell` returns a list of `MacroStep` (the *trace*) instead of an `Interpreter` instance.
- (Changed) The name of some attributes of an event in a tester story changes (e.g. *event* becomes *consumed_event*, *state* becomes *entered_state* or *exited_state* or *source_state* or *target_state*).
- (Removed) `Interpreter.trace`, as it can be easily obtained from `execute_once` or using `log_trace`.
- (Removed) `Interpreter.__init__` does not accept an `initial_time` parameter.
- (Fixed) Parallel state without children does not any more result into an infinite loop.

2.13.39 0.19.0 (2016-02-10)

- (Added) BDD can now output coverage data using `--coverage` command-line argument.
- (Changed) The YAML definition of a statechart must use *root state:* instead of *initial state:*.
- (Changed) When a contract is evaluated by a `PythonEvaluator`, `__old__.x` raises an `AttributeError` instead of a `KeyError` if `x` does not exist.

- (Changed) Behave is now called from Python instead of using a subprocess and thus allows debugging.

2.13.40 0.18.1 (2016-02-03)

- (Added) Support for behavior-driven-development using Behave.

2.13.41 0.17.3 (2016-01-29)

- (Added) An `io.text.export_to_tree` that returns a textual representation of the states.
- (Changed) `Statechart.rename_to` does not anymore raise `KeyError` but `exceptions.StatechartError`.
- (Changed) Wheel build should work on Windows

2.13.42 0.17.1 (2016-01-25)

Many backward incompatible changes in this update, especially if you used to work with `model`. The YAML format of a statechart also changed, look carefully at the changelog and the documentation.

- (Added) YAML: an history state can declare *on entry* and *on exit*.
- (Added) Statechart: new methods to manipulate transitions: `transitions_from`, `transitions_to`, `transitions_with`, `remove_transition` and `rotate_transition`.
- (Added) Statechart: new methods to manipulate states: `remove_state`, `rename_state`, `move_state`, `state_for`, `parent_for`, `children_for`.
- (Added) Steps: `__eq__` for `MacroStep` and `MicroStep`.
- (Added) Stories: `tell_by_step` method for a `Story`.
- (Added) Testing: `teststory_from_trace` generates a *step* event at the beginning of each step.
- (Added) Module: a new exceptions hierarchy (see `exceptions` module). The new exceptions are used in place of the old ones (`Warning`, `AssertionError` and `ValueError`).
- (Changed) YAML: uppermost *states:* should be replaced by *initial state:* and can contain at most one state.
- (Changed) YAML: uppermost *on entry:* should be replaced by *preamble:*.
- (Changed) YAML: initial memory of an history state should be specified using *memory* instead of *initial*.
- (Changed) YAML: contracts for a statechart must be declared on its root state.
- (Changed) Statechart: rename `StateChart` to `Statechart`.
- (Changed) Statechart: rename `events` to `events_for`.
- (Changed) Statechart: `states` attribute is now `Statechart.state_for` method.
- (Changed) Statechart: `register_state` is now `add_state`.
- (Changed) Statechart: `register_transition` is now `add_transition`.
- (Changed) Statechart: now defines a root state.
- (Changed) Statechart: checks done in `validate`.
- (Changed) Transition: `.event` is a string instead of an `Event` instance.
- (Changed) Transition: attributes `from_state` and `to_state` are renamed into `source` and `target`.

- (Changed) Event: `__eq__` takes `data` attribute into account.
- (Changed) Event: `event.foo` raises an `AttributeError` instead of a `KeyError` if `foo` is not defined.
- (Changed) State: `StateMixin.name` is now read-only (use `Statechart.rename_state`).
- (Changed) State: split `HistoryState` into a mixin `HistoryStateMixin` and two concrete subclasses, namely `ShallowHistoryState` and `DeepHistoryState`.
- (Changed) IO: Complete rewrite of `io.import_from_yaml` to load states before transitions. Parameter names have changed.
- (Changed) Module: adapt module hierarchy (no visible API change).
- (Changed) Module: expose module content through `__all__`.
- (Removed) Transition: `transitions` attribute on `TransitionStateMixin`, use `Statechart.transitions_for` instead.
- (Removed) State: `CompositeStateMixin.children`, use `Statechart.children_for` instead.

2.13.43 0.16.0 (2016-01-15)

- (Added) An `InternalEvent` subclass for `model.Event`.
- (Added) `Interpreter` now exposes its `statechart`.
- (Added) `Statechart.validate` checks that a targeted compound state declares an initial state.
- (Changed) `Interpreter.queue` does not accept anymore an `internal` parameter. Use an instance of `InternalEvent` instead (#20).
- (Fixed) `Story.story_from_trace` now ignores internal events (#19).
- (Fixed) Condition C3 in `Statechart.validate`.

2.13.44 0.15.0 (2016-01-12)

- (Changed) Rename `Interpreter.send` to `Interpreter.queue` (#18).
- (Changed) Rename `evaluator` module to `code`.

2.13.45 0.14.3 (2016-01-12)

- (Added) Changelog.
- (Fixed) Missing files in MANIFEST.in

2.14 API Reference

2.14.1 Module *bdd*

`sismic.bdd.`**`execute_bdd`**(*statechart*, *feature_filepaths*, ***, *step_filepaths=None*, *property_statecharts=None*, *interpreter_klass=<class 'sismic.interpreter.default.Interpreter'>*, *debug_on_error=False*, *behave_parameters=None*)

Execute BDD tests for a statechart.

Parameters

- **statechart** (Statechart) – statechart to test
- **feature_filepaths** (List[str]) – list of filepaths to feature files.
- **step_filepaths** (Optional[List[str]]) – list of filepaths to step definitions.
- **property_statecharts** (Optional[List[Statechart]]) – list of property statecharts
- **interpreter_klass** (Callable[[Statechart], Interpreter]) – a callable that accepts a statechart and returns an Interpreter
- **debug_on_error** (bool) – set to True to drop to (i)pdb in case of error.
- **behave_parameters** (Optional[List[str]]) – additional CLI parameters used by Behave (see http://behave.readthedocs.io/en/latest/behave.html#command-line-arguments)

Return type int

Returns exit code of behave CLI.

`sismic.bdd.`**`map_action`**(*step_text*, *existing_step_or_steps*)

Map new "given"/"when" steps to one or many existing one(s). Parameters are propagated to the original step(s) as well, as expected.

Examples:

- map_action('I open door', 'I send event open_door')
- map_action('Event {name} has to be sent', 'I send event {name}')
- map_action('I do two things', ['First thing to do', 'Second thing to do'])

Parameters

- **step_text** (str) – Text of the new step, without the "given" or "when" keyword.
- **existing_step_or_steps** (Union[str, List[str]]) – existing step, without the "given" or "when" keyword. Could be a list of steps.

Return type None

`sismic.bdd.`**`map_assertion`**(*step_text*, *existing_step_or_steps*)

Map a new "then" step to one or many existing one(s). Parameters are propagated to the original step(s) as well, as expected.

map_assertion('door is open', 'state door open is active') map_assertion('{x} seconds elapsed', 'I wait for {x} seconds') map_assertion('assert two things', ['first thing to assert', 'second thing to assert'])

Parameters

- **step_text** (str) – Text of the new step, without the "then" keyword.
- **existing_step_or_steps** (Union[str, List[str]]) – existing step, without "then" keyword. Could be a list of steps.

Return type None

2.14.2 Module *code*

class sismic.code.**Evaluator**(*interpreter=None*, ***, *initial_context=None*)

Bases: object

Abstract base class for any evaluator.

An instance of this class defines what can be done with piece of codes contained in a statechart (condition, action, etc.).

Notice that the execute_* methods are called at each step, even if there is no code to execute. This allows the evaluator to keep track of the states that are entered or exited, and of the transitions that are processed.

Parameters

- **interpreter** – the interpreter that will use this evaluator, is expected to be an *Interpreter* instance
- **initial_context** (Optional[Mapping[str, Any]]) – an optional dictionary to populate the context

context

The context of this evaluator. A context is a dict-like mapping between variables and values that is expected to be exposed when the code is evaluated.

Return type Mapping[str, Any]

evaluate_guard(*transition*, *event=None*)

Evaluate the guard for given transition.

Parameters

- **transition** (Transition) – the considered transition
- **event** (Optional[Event]) – instance of *Event* if any

Return type Optional[bool]

Returns truth value of *code*

evaluate_invariants(*obj*, *event=None*)

Evaluate the invariants for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterable[str]

Returns list of unsatisfied conditions

evaluate_postconditions(*obj*, *event=None*)

Evaluate the postconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterable[str]

Returns list of unsatisfied conditions

evaluate_preconditions(*obj*, *event=None*)

Evaluate the preconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterable[str]

Returns list of unsatisfied conditions

execute_action(*transition*, *event=None*)

Execute the action for given transition. This method is called for every transition that is processed, even those with no *action*.

Parameters

- **transition** (Transition) – the considered transition
- **event** (Optional[Event]) – instance of *Event* if any

Return type List[Event]

Returns a list of sent events

execute_on_entry(*state*)

Execute the on entry action for given state. This method is called for every state that is entered, even those with no *on_entry*.

Parameters **state** (StateMixin) – the considered state

Return type List[Event]

Returns a list of sent events

execute_on_exit(*state*)

Execute the on exit action for given state. This method is called for every state that is exited, even those with no *on_exit*.

Parameters **state** (StateMixin) – the considered state

Return type List[Event]

Returns a list of sent events

execute_statechart(*statechart*)

Execute the initial code of a statechart. This method is called at the very beginning of the execution.

Parameters **statechart** (Statechart) – statechart to consider

on_step_starts(*event=None*)

Called each time the interpreter starts a macro step.

Parameters **event** (Optional[Event]) – Optional processed event

Return type None

class sismic.code.**DummyEvaluator**(*interpreter=None*, *, *initial_context=None*)

Bases: sismic.code.evaluator.Evaluator

A dummy evaluator that does nothing and evaluates every condition to True.

context
The context of this evaluator. A context is a dict-like mapping between variables and values that is expected to be exposed when the code is evaluated.

evaluate_guard(*transition*, *event=None*)
Evaluate the guard for given transition.

Parameters

- **transition** (Transition) – the considered transition
- **event** (Optional[Event]) – instance of *Event* if any

Return type Optional[bool]

Returns truth value of *code*

evaluate_invariants(*obj*, *event=None*)
Evaluate the invariants for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterable[str]

Returns list of unsatisfied conditions

evaluate_postconditions(*obj*, *event=None*)
Evaluate the postconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterable[str]

Returns list of unsatisfied conditions

evaluate_preconditions(*obj*, *event=None*)
Evaluate the preconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterable[str]

Returns list of unsatisfied conditions

execute_action(*transition*, *event=None*)
Execute the action for given transition. This method is called for every transition that is processed, even those with no *action*.

Parameters

- **transition** (Transition) – the considered transition
- **event** (Optional[Event]) – instance of *Event* if any

Return type `List[Event]`

Returns a list of sent events

execute_on_entry(*state*)
Execute the on entry action for given state. This method is called for every state that is entered, even those with no *on_entry*.

Parameters **state** (`StateMixin`) – the considered state

Return type `List[Event]`

Returns a list of sent events

execute_on_exit(*state*)
Execute the on exit action for given state. This method is called for every state that is exited, even those with no *on_exit*.

Parameters **state** (`StateMixin`) – the considered state

Return type `List[Event]`

Returns a list of sent events

execute_statechart(*statechart*)
Execute the initial code of a statechart. This method is called at the very beginning of the execution.

Parameters **statechart** (`Statechart`) – statechart to consider

on_step_starts(*event=None*)
Called each time the interpreter starts a macro step.

Parameters **event** (`Optional[Event]`) – Optional processed event

Return type None

class sismic.code.**PythonEvaluator**(*interpreter=None*, *, *initial_context=None*)
Bases: `sismic.code.evaluator.Evaluator`

A code evaluator that understands Python.

Depending on the method that is called, the context can expose additional values:

- **On both code execution and code evaluation:**
  - A *time: float* value that represents the current time exposed by the interpreter.
  - An *active(name: str) -> bool* Boolean function that takes a state name and return *True* if and only if this state is currently active, ie. it is in the active configuration of the `Interpreter` instance that makes use of this evaluator.
- **On code execution:**
  - A *send(name: str, **kwargs) -> None* function that takes an event name and additional keyword parameters and raises an internal event with it.
  - If the code is related to a transition, the *event: Event* that fires the transition is exposed.
- **On guard or contract evaluation:**
  - If the code is related to a transition, the *event: Event* that fires the transition is exposed.
- **On guard or contract (except preconditions) evaluation:**
  - An *after(sec: float) -> bool* Boolean function that returns *True* if and only if the source state was entered more than *sec* seconds ago. The time is evaluated according to Interpreter's clock.

  - A *idle(sec: float) -> bool* Boolean function that returns *True* if and only if the source state did not fire a transition for more than *sec* ago. The time is evaluated according to Interpreter's clock.

- **On contract (except preconditions) evaluation:**
  - A variable *__old__* that has an attribute *x* for every *x* in the context when either the state was entered (if the condition involves a state) or the transition was processed (if the condition involves a transition). The value of *__old__.x* is a shallow copy of *x* at that time.

- **On contract evaluation:**
  - A *sent(name: str) -> bool* function that takes an event name and return True if an event with the same name was sent during the current step.
  - A *received(name: str) -> bool* function that takes an event name and return True if an event with the same name is currently processed in this step.

If an exception occurred while executing or evaluating a piece of code, it is propagated by the evaluator.

Parameters

- **`interpreter`** – the interpreter that will use this evaluator, is expected to be an *Interpreter* instance
- **`initial_context`** (`Optional[Mapping[str, Any]]`) – a dictionary that will be used as *__locals__*

`context`

The context of this evaluator. A context is a dict-like mapping between variables and values that is expected to be exposed when the code is evaluated.

Return type `Mapping[~KT, +VT_co]`

`evaluate_guard`(*transition*, *event=None*)

Evaluate the guard for given transition.

Parameters

- **`transition`** (`Transition`) – the considered transition
- **`event`** (`Optional[Event]`) – instance of *Event* if any

Return type `bool`

Returns truth value of *code*

`evaluate_invariants`(*obj*, *event=None*)

Evaluate the invariants for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **`obj`** – the considered state or transition
- **`event`** (`Optional[Event]`) – an optional *Event* instance, in the case of a transition

Return type `Iterator[str]`

Returns list of unsatisfied conditions

`evaluate_postconditions`(*obj*, *event=None*)

Evaluate the postconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **`obj`** – the considered state or transition

- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterator[str]

Returns list of unsatisfied conditions

evaluate_preconditions (*obj*, *event=None*)
Evaluate the preconditions for given object (either a *StateMixin* or a *Transition*) and return a list of conditions that are not satisfied.

Parameters

- **obj** – the considered state or transition
- **event** (Optional[Event]) – an optional *Event* instance, in the case of a transition

Return type Iterator[str]

Returns list of unsatisfied conditions

execute_action (*transition*, *event=None*)
Execute the action for given transition. This method is called for every transition that is processed, even those with no *action*.

Parameters

- **transition** (Transition) – the considered transition
- **event** (Optional[Event]) – instance of *Event* if any

Return type List[Event]

Returns a list of sent events

execute_on_entry (*state*)
Execute the on entry action for given state. This method is called for every state that is entered, even those with no *on_entry*.

Parameters **state** (StateMixin) – the considered state

Return type List[Event]

Returns a list of sent events

execute_on_exit (*state*)
Execute the on exit action for given state. This method is called for every state that is exited, even those with no *on_exit*.

Parameters **state** (StateMixin) – the considered state

Return type List[Event]

Returns a list of sent events

execute_statechart (*statechart*)
Execute the initial code of a statechart. This method is called at the very beginning of the execution.

Parameters **statechart** (Statechart) – statechart to consider

on_step_starts (*event=None*)
Called each time the interpreter starts a macro step.

Parameters **event** (Optional[Event]) – Optional processed event

Return type None

2.14.3 Module *exceptions*

exception sismic.exceptions.**CodeEvaluationError**

Bases: *sismic.exceptions.SismicError*

Base error for anything related to the evaluation of the code contained in a statechart.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**ConflictingTransitionsError**

Bases: *sismic.exceptions.ExecutionError*

When multiple conflicting (parallel) transitions can be processed at the same time.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**ContractError**(*configuration=None*, *step=None*, *obj=None*, *assertion=None*, *context=None*)

Bases: *sismic.exceptions.SismicError*

Base exception for situations in which a contract is not satisfied. All the parameters are optional, and are exposed to ease debug.

Parameters

- **configuration** – list of active states
- **step** – a *MicroStep* or *MacroStep* instance.
- **obj** – the object that is concerned by the assertion
- **assertion** – the assertion that failed
- **context** – the context in which the condition failed

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**ExecutionError**

Bases: *sismic.exceptions.SismicError*

Base error for anything related to the execution of a statechart.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**InvariantError**(*configuration=None*, *step=None*, *obj=None*, *assertion=None*, *context=None*)

Bases: *sismic.exceptions.ContractError*

An invariant is not satisfied.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**NonDeterminismError**

Bases: *sismic.exceptions.ExecutionError*

In case of non-determinism.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**PostconditionError**(*configuration=None*, *step=None*, *obj=None*, *assertion=None*, *context=None*)

Bases: *sismic.exceptions.ContractError*

A postcondition is not satisfied.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**PreconditionError**(*configuration=None*, *step=None*, *obj=None*, *assertion=None*, *context=None*)

Bases: *sismic.exceptions.ContractError*

A precondition is not satisfied.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**PropertyStatechartError**(*property_statechart*, *configuration*, *step*, *context*)

Bases: *sismic.exceptions.SismicError*

Raised when a property statechart reaches a final state.

Parameters

- **property_statechart** – the property statechart that reaches a final state
- **configuration** – list of active states
- **step** – latest executed macro step
- **context** – the context in which the condition failed

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**SismicError**

Bases: Exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception sismic.exceptions.**StatechartError**

Bases: *sismic.exceptions.SismicError*

Base error for anything that is related to a statechart.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.14.4 Module *helpers*

sismic.helpers.**log_trace**(*interpreter*)

Return a list that will be populated by each value returned by the *execute_once* method of given interpreter.

Parameters **interpreter** (Interpreter) – an *Interpreter* instance

Return type List[MacroStep]

Returns a list of *MacroStep*

`sismic.helpers.`**`run_in_background`**(*interpreter*, *delay=0.05*, *callback=None*)

Run given interpreter in background. The time is updated according to *time.time() - starttime*. The interpreter is ran until it reaches a final configuration. You can manually stop the thread using the added *stop* of the returned Thread object. This is for convenience only and should be avoided, because a call to *stop* puts the interpreter in an empty (and thus final) configuration, without properly leaving the active states.

Parameters

- **`interpreter`** (`Interpreter`) – an interpreter
- **`delay`** (`float`) – delay between each call to *execute()*
- **`callback`** (`Optional[Callable[[List[MacroStep]], Any]]`) – a function that accepts the result of *execute*.

Return type `Thread`

Returns started thread (instance of *threading.Thread*)

`sismic.helpers.`**`coverage_from_trace`**(*trace*)

Given a list of macro steps considered as the trace of a statechart execution, return *Counter* objects that counts the states that were entered, the states that were exited and the transitions that were processed.

Parameters **`trace`** (`List[MacroStep]`) – A list of macro steps

Return type `Mapping[str, Counter]`

Returns A dict whose keys are "entered states", "exited states" and "processed transitions" and whose values are Counter object.

2.14.5 Module *interpreter*

class `sismic.interpreter.`**`Interpreter`**(*statechart*, ***, *evaluator_klass=<class 'sismic.code.python.PythonEvaluator'>*, *initial_context=None*, *ignore_contract=False*)

Bases: `object`

A discrete interpreter that executes a statechart according to a semantic close to SCXML.

Parameters

- **`statechart`** (`Statechart`) – statechart to interpret
- **`evaluator_klass`** (`Callable[..., Evaluator]`) – An optional callable (eg. a class) that takes an interpreter and an optional initial context as input and return an *Evaluator* instance that will be used to initialize the interpreter. By default, the *PythonEvaluator* class will be used.
- **`initial_context`** (`Optional[Mapping[str, Any]]`) – an optional initial context that will be provided to the evaluator. By default, an empty context is provided
- **`ignore_contract`** (`bool`) – set to True to ignore contract checking during the execution.

`bind`(*interpreter_or_callable*)

Bind an interpreter or a callable to the current interpreter. Each time an internal event is sent by this interpreter, any bound object will be called with the same event. If *interpreter_or_callable* is an *Interpreter* instance, its *queue* method is called. This is, if *i1* and *i2* are interpreters, *i1.bind(i2)* is equivalent to *i1.bind(i2.queue)*.

Parameters **`interpreter_or_callable`** (`Union[Interpreter, Callable[[Event], Any]]`) – interpreter or callable to bind

Return type None

Returns *self* so it can be chained

bind_property_statechart(*statechart_or_interpreter*)
Bind a property statechart to the current interpreter. A property statechart receives meta-events from the current interpreter depending on what happens:

- *step started*: when a macro step starts.
- *step ended*: when a macro step ends.
- *event consumed*: when an event is consumed. The consumed event is exposed through the `event` attribute.
- *event sent*: when an event is sent. The sent event is exposed through the `event` attribute.
- *state exited*: when a state is exited. The exited state is exposed through the `state` attribute.
- *state entered*: when a state is entered. The entered state is exposed through the `state` attribute.
- *transition processed*: when a transition is processed. The source state, target state and the event are exposed respectively through the `source`, `target` and `event` attribute.

The internal clock of all property statecharts will be synced with the one of the current interpreter. As soon as a property statechart reaches a final state, a `PropertyStatechartError` will be raised, implicitly meaning that the property expressed by the corresponding property statechart is not satisfied.

Parameters **statechart_or_interpreter** (`Union[Statechart, Interpreter]`) – A property statechart or an interpreter of a property statechart.

Return type None

configuration
List of active states names, ordered by depth. Ties are broken according to the lexicographic order on the state name.

Return type `List[str]`

context
The context of execution.

Return type `Mapping[str, Any]`

execute(*max_steps=-1*)
Repeatedly calls *execute_once* and return a list containing the returned values of *execute_once*.

Notice that this does NOT return an iterator but computes the whole list first before returning it.

Parameters **max_steps** (`int`) – An upper bound on the number steps that are computed and returned. Default is -1, no limit. Set to a positive integer to avoid infinite loops in the statechart execution.

Return type `List[MacroStep]`

Returns A list of *MacroStep* instances

execute_once()
Select transitions that can be fired based on available queued events, process them and stabilize the interpreter. When multiple transitions are selected, they are atomically processed: states are exited, transition is processed, states are entered, statechart is stabilized and only after that, the next transition is processed.

Return type `Optional[MacroStep]`

Returns a macro step or *None* if nothing happened

final
Boolean indicating whether this interpreter is in a final configuration.

Return type `bool`

queue (*event_or_name*, **events_or_names*)
Queue one or more events to the interpreter external queue.

Parameters

- **event_or_name** (Union[str, Event]) – an *Event* instance, or the name of an event.
- **events_or_names** (Union[str, Event]) – additional *Event* instances, or names of events.

Return type `Interpreter`

Returns *self* so it can be chained.

statechart
Embedded statechart

Return type `Statechart`

time
Time value (in seconds) for the internal clock

Return type `float`

class sismic.interpreter.**Event** (*name*, ***additional_parameters*)
Bases: `object`

Simple event with a name and (optionally) some data. Unless the attribute already exists, each key from *data* is exposed as an attribute of this class.

The list of defined attributes can be obtained using *dir(event)*.

Parameters

- **name** (str) – Name of the event
- **data** – additional data (mapping, dict-like)

class sismic.interpreter.**InternalEvent** (*name*, ***additional_parameters*)
Bases: `sismic.model.events.Event`

Subclass of Event that represents an internal event.

class sismic.interpreter.**MetaEvent** (*name*, ***additional_parameters*)
Bases: `sismic.model.events.Event`

Subclass of Event that represents a MetaEvent, as used in property statecharts.

2.14.6 Module *io*

sismic.io.**import_from_yaml** (*text=None*, *filepath=None*, ***, *ignore_schema=False*, *ignore_validation=False*)
Import a statechart from a YAML representation (first argument) or a YAML file (filepath argument).

Unless specified, the structure contained in the YAML is validated against a predefined schema (see *sismic.io.SCHEMA*), and the resulting statechart is validated using its *validate()* method.

Parameters

- **text** (Optional[Iterable[str]]) – A YAML text. If not provided, filepath argument has to be provided.
- **filepath** (Optional[str]) – A path to a YAML file.
- **ignore_schema** (bool) – set to *True* to disable yaml validation.
- **ignore_validation** (bool) – set to *True* to disable statechart validation.

Return type Statechart

Returns a *Statechart* instance

sismic.io.**export_to_yaml**(*statechart*, *filepath=None*)

Export given *Statechart* instance to YAML. Its YAML representation is returned by this function. Automatically save the output to filepath, if provided.

Parameters

- **statechart** (Statechart) – statechart to export
- **filepath** (Optional[str]) – save output to given filepath, if provided

Return type str

Returns A textual YAML representation

sismic.io.**export_to_plantuml**(*statechart*, *filepath=None*, ***, *based_on=None*, *based_on_filepath=None*, *statechart_name=True*, *statechart_description=False*, *statechart_preamble=False*, *state_contracts=False*, *state_action=True*, *transition_contracts=False*, *transition_action=True*)

Export given statechart to plantUML (see http://plantuml/plantuml). If a filepath is provided, also save the output to this file.

Due to the way statecharts are representing, and due to the presence of features that are specific to Sismic, the resulting statechart representation does not include all the informations. For example, final states and history states won't have name, actions and contracts.

If a previously exported representation for the statechart is provided, either as text (based_on parameter) or as a filepath (based_on_filepath parameter), it will attempt to reuse the modifications made to the transitions (their direction and length).

Parameters

- **statechart** (Statechart) – statechart to export
- **filepath** (Optional[str]) – save output to given filepath, if provided
- **based_on** (Optional[str]) – existing representation of the statechart in PlantUML
- **based_on_filepath** (Optional[str]) – filepath to an existing representation of the statechart in PlantUML
- **statechart_name** (bool) – include the name of the statechart
- **statechart_description** (bool) – include the description of the statechart
- **statechart_preamble** (bool) – include the preamble of the statechart
- **state_contracts** (bool) – include state contracts
- **state_action** (bool) – include state actions (on entry, on exit and internal transitions)
- **transition_contracts** (bool) – include transition contracts
- **transition_action** (bool) – include actions on transition

Return type `str`

Returns textual representation using plantuml

2.14.7 Module *model*

class `sismic.model.`**`ActionStateMixin`**(*on_entry=None*, *on_exit=None*)

Bases: `object`

State that can define actions on entry and on exit.

Parameters

- **`on_entry`** (`Optional[str]`) – code to execute when state is entered
- **`on_exit`** (`Optional[str]`) – code to execute when state is exited

class `sismic.model.`**`BasicState`**(*name*, *on_entry=None*, *on_exit=None*)

Bases: `sismic.model.elements.ContractMixin`, `sismic.model.elements.StateMixin`, `sismic.model.elements.ActionStateMixin`, `sismic.model.elements.TransitionStateMixin`

A basic state, with a name, transitions, actions, etc. but no child state.

Parameters

- **`name`** (`str`) – name of this state
- **`on_entry`** (`Optional[str]`) – code to execute when state is entered
- **`on_exit`** (`Optional[str]`) – code to execute when state is exited

class `sismic.model.`**`CompositeStateMixin`**

Bases: `object`

Composite state can have children states.

class `sismic.model.`**`CompoundState`**(*name*, *initial=None*, *on_entry=None*, *on_exit=None*)

Bases: `sismic.model.elements.ContractMixin`, `sismic.model.elements.StateMixin`, `sismic.model.elements.ActionStateMixin`, `sismic.model.elements.TransitionStateMixin`, `sismic.model.elements.CompositeStateMixin`

Compound states must have children states.

Parameters

- **`name`** (`str`) – name of this state
- **`initial`** (`Optional[str]`) – name of the initial state
- **`on_entry`** (`Optional[str]`) – code to execute when state is entered
- **`on_exit`** (`Optional[str]`) – code to execute when state is exited

class `sismic.model.`**`ContractMixin`**

Bases: `object`

Mixin with a contract: preconditions, postconditions and invariants.

class `sismic.model.`**`DeepHistoryState`**(*name*, *on_entry=None*, *on_exit=None*, *memory=None*)

Bases: `sismic.model.elements.ContractMixin`, `sismic.model.elements.StateMixin`, `sismic.model.elements.ActionStateMixin`, `sismic.model.elements.HistoryStateMixin`

A deep history state resumes the execution of its parent, and of every nested active states in its parent.

Parameters

- **name** (str) – name of this state
- **on_entry** (Optional[str]) – code to execute when state is entered
- **on_exit** (Optional[str]) – code to execute when state is exited
- **memory** (Optional[str]) – name of the initial state

class sismic.model.**Event**(*name*, ***additional_parameters*)

Bases: object

Simple event with a name and (optionally) some data. Unless the attribute already exists, each key from *data* is exposed as an attribute of this class.

The list of defined attributes can be obtained using *dir(event)*.

Parameters

- **name** (str) – Name of the event
- **data** – additional data (mapping, dict-like)

class sismic.model.**FinalState**(*name*, *on_entry=None*, *on_exit=None*)

Bases: sismic.model.elements.ContractMixin, sismic.model.elements.StateMixin, sismic.model.elements.ActionStateMixin

Final state has NO transition and is used to detect state machine termination.

Parameters

- **name** (str) – name of this state
- **on_entry** (Optional[str]) – code to execute when state is entered
- **on_exit** (Optional[str]) – code to execute when state is exited

class sismic.model.**HistoryStateMixin**(*memory=None*)

Bases: object

History state has a memory that can be resumed.

Parameters **memory** (Optional[str]) – name of the initial state

class sismic.model.**InternalEvent**(*name*, ***additional_parameters*)

Bases: sismic.model.events.Event

Subclass of Event that represents an internal event.

class sismic.model.**MacroStep**(*time*, *steps*)

Bases: object

A macro step is a list of micro steps.

Parameters

- **time** (float) – the time at which this step was executed
- **steps** (List[MicroStep]) – a list of *MicroStep* instances

entered_states

List of the states names that were entered.

Return type List[str]

event

Event (or *None*) that was consumed.

Return type `Optional[Event]`

exited_states
List of the states names that were exited.

Return type `List[str]`

sent_events
List of events that were sent during this step.

Return type `List[Event]`

steps
List of micro steps

Return type `List[MicroStep]`

time
Time at which this step was executed.

Return type `float`

transitions
A (possibly empty) list of transitions that were triggered.

Return type `List[Transition]`

class `sismic.model.`**MetaEvent**(*name*, ***additional_parameters*)
Bases: `sismic.model.events.Event`

Subclass of Event that represents a MetaEvent, as used in property statecharts.

class `sismic.model.`**MicroStep**(*event=None*, *transition=None*, *entered_states=None*, *exited_states=None*, *sent_events=None*)
Bases: `object`

Create a micro step.

A step consider *event*, takes a *transition* and results in a list of *entered_states* and a list of *exited_states*. Order in the two lists is REALLY important!

Parameters

- **event** (`Optional[Event]`) – Event or None in case of eventless transition
- **transition** (`Optional[Transition]`) – a *Transition* or None if no processed transition
- **entered_states** (`Optional[List[str]]`) – possibly empty list of entered states
- **exited_states** (`Optional[List[str]]`) – possibly empty list of exited states
- **sent_events** (`Optional[List[Event]]`) – a possibly empty list of events that are sent during the step

class `sismic.model.`**OrthogonalState**(*name*, *on_entry=None*, *on_exit=None*)
Bases: `sismic.model.elements.ContractMixin`, `sismic.model.elements.StateMixin`, `sismic.model.elements.ActionStateMixin`, `sismic.model.elements.TransitionStateMixin`, `sismic.model.elements.CompositeStateMixin`

Orthogonal states run their children simultaneously.

Parameters

- **name** (`str`) – name of this state
- **on_entry** (`Optional[str]`) – code to execute when state is entered

- **on_exit** (Optional[str]) – code to execute when state is exited

class sismic.model.**ShallowHistoryState**(*name*, *on_entry=None*, *on_exit=None*, *memory=None*)

Bases: sismic.model.elements.ContractMixin, sismic.model.elements.StateMixin, sismic.model.elements.ActionStateMixin, sismic.model.elements.HistoryStateMixin

A shallow history state resumes the execution of its parent. It activates the latest visited state of its parent.

Parameters

- **name** (str) – name of this state
- **on_entry** (Optional[str]) – code to execute when state is entered
- **on_exit** (Optional[str]) – code to execute when state is exited
- **memory** (Optional[str]) – name of the initial state

class sismic.model.**StateMixin**(*name*)

Bases: object

State element with a name.

Parameters **name** (str) – name of the state

class sismic.model.**Statechart**(*name*, *description=None*, *preamble=None*)

Bases: object

Python structure for a statechart

Parameters

- **name** (str) – Name of this statechart
- **description** (Optional[str]) – optional description
- **preamble** (Optional[str]) – code to execute to bootstrap the statechart

add_state(*state*, *parent*)

Add given state (a *StateMixin* instance) on given parent (its name as an *str*). If given state should be use as a root state, set *parent* to None.

Parameters

- **state** (StateMixin) – state to add
- **parent** (Optional[str]) – name of its parent, or None

Raises *StatechartError* –

Return type None

add_transition(*transition*)

Register given transition and register it on the source state

Parameters **transition** (Transition) – transition to add

Raises *StatechartError* –

Return type None

ancestors_for(*name*)

Return an ordered list of ancestors for the given state. Ancestors are ordered by decreasing depth.

Parameters **name** (str) – name of the state

Return type List[str]

Returns state's ancestors

Raises *StatechartError* – if state does not exist

children_for (*name*)

Return the names of the children of the given state.

Parameters **name** (str) – a state name

Return type List[str]

Returns a (possibly empty) list of children

Raises *StatechartError* – if state does not exist

copy_from_statechart (*statechart*, ***, *source*, *replace*, *renaming_func=<function Statechart.<lambda>>*)

Copy (a part of) given *statechart* into current one.

Copy *source* state, all its descendants and all involved transitions from *statechart* into current statechart. The *source* state will override *replace* state (but will be renamed to *replace*), and all its descendants in *statechart* will be copied into current statechart. All the transitions that are involved in the process must be fully contained in *source* state (ie. for all transition T: S->T, if S (resp. T) is a descendant-or-self of *source*, then T (resp. S) must be a descendant-or-self of *source*).

If necessary, callable *renaming_func* can be provided. This function should accept a (state) name and return a (new state) name. Use *renaming_func* to avoid conflicting names in target statechart.

Parameters

- **statechart** (Statechart) – Source statechart from which states will be copied.
- **source** (str) – Name of the source state.
- **replace** (str) – Name of the target state. Should refer to a StateMixin with no child.
- **renaming_func** (Callable[[str], str]) – Optional callable to resolve conflicting names.

Return type None

depth_for (*name*)

Return the depth of given state (1-indexed).

Parameters **name** (str) – name of the state

Return type int

Returns state depth

Raises *StatechartError* – if state does not exist

descendants_for (*name*)

Return an ordered list of descendants for the given state. Descendants are ordered by increasing depth.

Parameters **name** (str) – name of the state

Return type List[str]

Returns state's descendants

Raises *StatechartError* – if state does not exist

events_for (*name_or_names=None*)

Return a list containing the name of every event that guards a transition in this statechart.

If *name_or_names* is specified, it must be the name of a state (or a list of such names). Only transitions that have a source state from this list will be considered. By default, the list contains all the states.

Parameters **name_or_names** (Union[str, List[str], None]) – *None*, a state name or a list of state names.

Return type List[str]

Returns A list of event names

leaf_for(*names*)
Return the leaves of *names*.

Considering the list of states names in *names*, return a list containing each element of *names* such that this element has no descendant in *names*.

Parameters **names** (Iterable[str]) – a list of state names

Return type List[str]

Returns the names of the leaves in *names*

Raises ***StatechartError*** – if a state does not exist

least_common_ancestor(*name_first*, *name_second*)
Return the deepest common ancestor for *s1* and *s2*, or *None* if there is no common ancestor except root (top-level) state.

Parameters

- **name_first** (str) – name of first state
- **name_second** (str) – name of second state

Return type Optional[str]

Returns name of deepest common ancestor or *None*

Raises ***StatechartError*** – if state does not exist

move_state(*name*, *new_parent*)
Move given state (and its children) such that its new parent is *new_parent*.

Notice that a state cannot be moved inside itself or inside one of its descendants. If the state to move is the target of an *initial* or *memory* property of its parent, this property will be set to None. The same occurs if given state is an history state.

Parameters

- **name** (str) – name of the state to move
- **new_parent** (str) – name of the new parent

Return type None

parent_for(*name*)
Return the name of the parent of given state name.

Parameters **name** (str) – a state name

Return type str

Returns its parent name, or None.

Raises ***StatechartError*** – if state does not exist

preamble
Preamble code

remove_state (*name*)
Remove given state.

The transitions that involve this state will also be removed. If the state is the target of an *initial* or *memory* property, their value will be set to None. If the state has children, they will be removed too.

Parameters **name** (str) – name of a state

Raises *StatechartError* –

Return type None

remove_transition (*transition*)
Remove given transitions.

Parameters **transition** (Transition) – a *Transition* instance

Raises *StatechartError* – if transition is not registered

Return type None

rename_state (*old_name*, *new_name*)
Change state name, and adapt transitions, initial state, memory, etc.

Parameters

- **old_name** (str) – old name of the state
- **new_name** (str) – new name of the state

Return type None

root
Root state name

Return type Optional[str]

rotate_transition (*transition*, *new_source=''*, *new_target=''*)
Rotate given transition.

You MUST specify either *new_source* (a valid state name) or *new_target* (a valid state name or None) or both.

Parameters

- **transition** (Transition) – a *Transition* instance
- **new_source** (str) – a state name
- **new_target** (Optional[str]) – a state name or None

Raises *StatechartError* – if given transition or a given state does not exist.

Return type None

state_for (*name*)
Return the state instance that has given name.

Parameters **name** (str) – a state name

Return type StateMixin

Returns a *StateMixin* that has the same name or None

Raises *StatechartError* – if state does not exist

states
List of state names in lexicographic order.

transitions
List of available transitions

transitions_from(*source*)
Return the list of transitions whose source is given name.

Parameters **source** (`str`) – name of source state

Return type `List[Transition]`

Returns a list of *Transition* instances

Raises ***StatechartError*** – if state does not exist

transitions_to(*target*)
Return the list of transitions whose target is given name. Internal transitions are returned too.

Parameters **target** (`str`) – name of target state

Return type `List[Transition]`

Returns a list of *Transition* instances

Raises ***StatechartError*** – if state does not exist

transitions_with(*event*)
Return the list of transitions that can be triggered by given event name.

Parameters **event** (`str`) – name of the event

Return type `List[Transition]`

Returns a list of *Transition* instances

validate()
Checks that every *CompoundState*'s initial state refer to one of its children Checks that every *HistoryStateMixin*'s memory refer to one of its parent's children

Return type `bool`

Returns True

Raises ***StatechartError*** –

class sismic.model.**Transition**(*source*, *target=None*, *event=None*, *guard=None*, *action=None*)
Bases: `sismic.model.elements.ContractMixin`

Represent a transition from a source state to a target state.

A transition can be eventless (no event) or internal (no target). A condition (code as string) can be specified as a guard.

Parameters

- **source** (`str`) – name of the source state
- **target** (`Optional[str]`) – name of the target state (if transition is not internal)
- **event** (`Optional[str]`) – event name (if any)
- **guard** (`Optional[str]`) – condition as code (if any)
- **action** (`Optional[str]`) – action as code (if any)

eventless
Boolean indicating whether this transition is an eventless transition.

internal
Boolean indicating whether this transition is an internal transition.

class sismic.model.**TransitionStateMixin**
Bases: object

A simple state can host transitions

CHAPTER 3

Credits

The Sismic library for Python is mainly developed by Alexandre Decan at the University of Mons with the help of many contributors.

Sismic is released publicly under the GNU Lesser General Public Licence version 3.0 (LGPLv3).

The source code is available on GitHub: https://github.com/AlexandreDecan/sismic

Use GitHub's integrated services to contribute suggestions and feature requests for this library or to report bugs.

You can cite Sismic using:

```
@software{sismic,
  author = {{Alexandre Decan}},
  title = {Sismic Interactive Statechart Model Interpreter and Checker},
  url = {https://github.com/AlexandreDecan/sismic},
}
```

Python Module Index

s

Index

Symbols

A

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

V

W